TURING 图灵程序设计丛书

20年后、100年后的编程语言会是什么样？

代码的未来



[日] 松本行弘◎著
日经Linux◎编　周自恒◎译

Ruby之父
剖析云计算、大数据时代下的技术

- Lisp会是未来的发展趋势吗？
- Go和Dart能取代C和JavaScript吗？
- 关系型数据库已经走到穷途末路了吗？

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。



作者

松本行弘

Ruby语言发明者，亦是亚洲首屈一指的编程语言发明者。现兼任网络应用通信研究所（NaCl）研究员、乐天技术研究所研究员、Heroku首席架构师等。昵称“Matz”。讨厌东京，喜欢温泉。



译者

周自恒

IT、编程爱好者，技术宅，初中时曾在NOI（国家信息学奥赛）天津赛区获一等奖，大学毕业后曾任IT咨询顾问，精通英语和日语，译著有《30天自制操作系统》、《大数据的冲击》、《Android应用开发入门》。

图灵程序设计丛书

代码的未来



[日] 松本行弘◎著
日经Linux◎编　周自恒◎译

人民邮电出版社
北　京

图书在版编目（CIP）数据

代码的未来 / （日）松本行弘著 ；周自恒译. -- 北京 ：人民邮电出版社，2013.5
（图灵程序设计丛书）
ISBN 978-7-115-31751-3

Ⅰ. ①代… Ⅱ. ①松… ②周… Ⅲ. ①程序语言 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第090707号

内 容 提 要

本书是 Ruby 之父松本行弘的又一力作。作者对云计算、大数据时代下的各种编程语言以及相关技术进行了剖析，并对编程语言的未来发展趋势做出预测，内容涉及 Go、VoltDB、node.js、CoffeeScript、Dart、MongoDB、摩尔定律、编程语言、多核、NoSQL 等当今备受关注的话题。

本书面向各层次程序设计人员和编程爱好者，也可供相关技术人员参考。

图灵程序设计丛书
代码的未来

◆ 著　　　　[日] 松本行弘
编　　　　日经Linux
译　　　　周自恒
责任编辑　乐 馨
执行编辑　金松月
责任印制　焦志炜
原版封面设计　石田昌治/株式会社マップス
（MASAHARU ISHIDA/MAPS Co.,Ltd.）

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：23
字数：544千字　　　　2013年 5 月第 1 版
印数：1－5 000册　　　2013年 5 月北京第 1 次印刷
著作权合同登记号　图字：01-2013-1117号
ISBN 978-7-115-31751-3

定价：79.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

译 者 序

依靠其简洁、优雅的语言特色，以及 Rails 等开发框架的成功，Ruby 在 Web 开发领域早已成为一种人气颇高的动态脚本语言。然而，当今世界上流行的编程语言中，只有 Ruby 来自亚洲，作为 Ruby 语言的发明者，松本行弘（Matz）表示自己常因此而感到孤独。

作为这本书的译者，2012 年 11 月借中国 Ruby 大会的机会，我有幸以图灵特派记者的身份对 Matz 进行了一次专访①。穿着 UNIQLO 的格子衬衫，充满技术宅范儿的 Matz，平时看起来不苟言笑，谈起技术话题来就好像打开了话匣子一般滔滔不绝，在 Twitter 上的发言也相当活跃。在访谈中，Matz 谈到了 Ruby 的发展方向，他希望 Ruby 能够在 Web 开发之外的领域（科学计算、高性能计算和嵌入式系统）有更多的发展，同时他也希望中国的程序员们能够积极为开源社区做出贡献，努力成为能够影响世界的工程师。

Matz 一直称自己是一个普通的程序员，创造 Ruby 只不过是他编程生涯中的一小部分。无论是以“资深 UNIX 程序员”的身份，还是“Ruby 之父”的身份，Matz 都有足够的资格对现今的编程语言和技术品头论足；另一方面，计算机技术的发展可谓日新月异，Matz 认为有必要从过去到未来，以发展的眼光来看待这些技术的演进。用资深程序员的视角和发展的眼光来剖析技术，这就是 Matz 笔下的《代码的未来》。

在这本书中，Matz 将和大家一起探讨丰富多彩的技术话题，并对编程语言的未来发展趋势做出自己的预测。像 Lisp 这样拥有最简核心的函数型语言真的会是未来的发展趋势吗？垃圾回收、闭包、高阶函数、元编程等编程语言中的要素是如何发展出来的？ Google 为什么要开发 Go 和 Dart，它们能取代 C 语言和 JavaScript 吗？大数据时代经常提到的 Hadoop、MapReduce、NoSQL 等名词到底是什么意思？关系型数据库真的已经走到穷途末路了吗？要充分运用多核心和分布式环境，在软件层面需要做出怎样的应对，又有哪些技术可以使用？如果你对上面这些话题感兴趣，无论心中是否已经有了自己的答案，都可以看一看来自 Matz 的解读。

① 访谈内容参见图灵社区：http://www.ituring.com.cn/article/17487。

和《松本行弘的程序世界》一样，这本书也是 Matz 在《日经 Linux》杂志连载的专栏文章的一个合集，书中选取的文章之间有近四年的时间跨度，且章节的安排也和原稿写作的时间顺序有所不同。不了解这个背景的读者，可能会被书中一些貌似前后重复或者“穿越”的地方搞得一头雾水——少安毋躁，这不是 bug。相比《松本行弘的程序世界》的 14 个主题来说，这本书的主题更加集中和深入，而不变的是，话题依然丰富，观点依然犀利，内容依然扎实，读起来畅快淋漓。

最后，感谢 Matz 在本书翻译过程中所给予的帮助和指导，感谢图灵公司各位编辑的辛苦工作，希望每位读者都能够从中有所收获。

周自恒

2013 年 3 月于上海

中文版序

人类的力量是有限的，无法完全通晓未来，因此我们并不能确切地知道明天、明年究竟会发生什么事。

不过，仅就技术来说，一夜之间就冒出个新东西，这样的情况是非常罕见的，而大多数新技术都是沿着从过去到现在的技术轨迹逐步发展起来的。在 IT 的世界中，这样的倾向尤其显著。

《代码的未来》综述了我当前掌握的 IT 趋势，书中就摩尔定律、编程语言、多核、NoSQL 等在未来几年中将备受关注的领域，介绍了相关的现状和基础知识。

当然，没人知道书中涉及的这些技术在更久远的未来是否还依然有用，但至少在不远的将来，它们应该是非常值得关注的技术。这些内容可以成为学习新技术的基础，对于想要成为优秀工程师、程序员的各位读者来说，这样的基础则能够成为生存竞争中的有力武器。

也许还有一些读者并非专职的程序员，但我认为本书同样值得他们一看。所谓技术，就是用来解决现实问题的手段。与现实问题展开的这场拉锯战，本身就是一件非常刺激和快乐的事，而这份快乐，也正是带动未来创新的源动力。

互联网和开源降低了参与创新的门槛。即便没有高学历，即便不属于任何一家企业，只要有技术和点子就有机会。可以想象，未来的创新就应该是这样。就 IT 方面来说，我认为大多数的创新应该都不外乎是本书介绍的这些技术的延伸。

有人说 21 世纪是亚洲的世纪。作为一个亚洲人，我开发的 Ruby 语言已经在全世界获得了广泛的应用，这也许从某种程度上印证了这种说法。这本书中包含了我的一些思考和见解，如果它能够对亚洲（恐怕应该是吧）各位读者的创新有所帮助，我会感到荣幸之至。

最后，希望中国的各位读者能够从本书中获益。

松本行弘

2013 年 4 月

前　言

本书是在《日经 Linux》上连载的《松本行弘：技术的剖析》（2009 年 6 月号～ 2012 年 6 月号）各期内容的合集。

老实说，写文章这件事很是让我头疼。我认为自己的本职工作是程序员，而不是作家。每个月构思一个主题、查阅资料、编写示例程序，然后再写成文章，这件事对我来说真是个负担。时间被占用，拖累了本职工作不说，截稿日前夕还得承受压力。因此那一阵子经常会感到无比焦虑。

话虽如此，但这件事也并非一无是处。在构思文章主题的时候，需要放眼于日常工作以外的世界，这样便拓宽了视野。其实，我本来也并不是那么讨厌写文章。说起来，在学生时代我成绩最好的科目还是语文和英语呢，而最差的科目则是数学。

因为是给杂志社供稿，所以我每个月都是选择当时那个时间点上比较热门的、能够引起我的兴趣的话题来写，并没有考虑到主题的连贯性。不过，借着编辑成书的机会回过头来看看以前连载的文章，和编辑讨论之后，头脑中便一下子浮现出“未来”这个关键词。连载中的每一篇文章原本都是独立的，但它们中的大多数都体现了“从过去到未来”、“应对即将到来的未来”这样的主题。作为这些文章的作者，我自己也感到颇为意外。

毋庸置疑，IT 技术正在创造着我们的现在和未来。无论是专业人士，还是业余爱好者，像我们这样的 IT 技术人，可以说是会最早与未来遭遇的“人种”吧。正是为了这些人，我才将《技术的剖析》这个专题连载至今。这些连载能浮现出“未来”这个共同的关键词，虽说事先没有预料到，但从某种意义上来说，也许是水到渠成自然而然的结果。

然而，IT 技术人的真正价值应该并非只有“最早与未来遭遇”而已，我们不仅要能够及早触及未来，还应该拥有自己创造未来的力量——创造出比这本书所预见的未来还要更加美好的未来。

松本行弘

2012 年 4 月

于樱花盛开的松江市

第1章 编程的时间和空间

1.1　编程的本质 3

编程的本质是思考 / 4
创造世界的乐趣 / 4
快速提高的性能改变了社会 / 5
以不变应万变 / 8
摩尔定律的局限 / 9
社会变化与编程 / 10

1.2　未来预测 13

科学的未来预测 / 14
IT 未来预测 / 14
极限未来预测 / 16
从价格看未来 / 16
从性能看未来 / 17
从容量看未来 / 18
从带宽看未来 / 19
小结 / 20

第2章 编程语言的过去、现在和未来

2.1　编程语言的世界 23

被历史埋没的先驱 / 25
编程语言的历史 / 26
编程语言的进化方向 / 31
未来的编程语言 / 33
20 年后的编程语言 / 34
学生们的想象 / 35

2.2　DSL（特定领域语言） 36

外部 DSL / 37
内部 DSL / 38
DSL 的优势 / 39
DSL 的定义 / 39
适合内部 DSL 的语言 / 40
外部 DSL 实例 / 42
DSL 设计的构成要素 / 43
Sinatra / 46
小结 / 47

2.3　元编程 48

Meta, Reflection / 48
类对象 / 51
类的操作 / 52
Lisp / 53
数据和程序 / 54
Lisp 程序 / 56
宏 / 56
宏的功与过 / 57
元编程的可能性与危险性 / 59
小结 / 60

2.4 内存管理......61

看似无限的内存 / 61
GC 的三种基本方式 / 62
术语定义 / 62
标记清除方式 / 63
复制收集方式 / 64
引用计数方式 / 65
引用计数方式的缺点 / 65
进一步改良的应用方式 / 66
分代回收 / 66
对来自老生代的引用进行记录 / 67
增量回收 / 68
并行回收 / 69
GC 大统一理论 / 69

2.5 异常处理......71

“一定没问题的” / 71
用特殊返回值表示错误 / 72
容易忽略错误处理 / 72
Ruby 中的异常处理 / 73
产生异常 / 74
更高级的异常处理 / 75
Ruby 中的后处理保证 / 76
其他语言中的异常处理 / 77
Java 的检查型异常 / 77
Icon 的异常和真假值 / 78
Eiffel 的 Design by Contract / 80
异常与错误值 / 80
小结 / 81

2.6 闭包......82

函数对象 / 82
高阶函数 / 83
用函数参数提高通用性 / 84
函数指针的局限 / 85
作用域：变量可见范围 / 87
生存周期：变量的存在范围 / 88
闭包与面向对象 / 89
Ruby 的函数对象 / 89
Ruby 与 JavaScript 的区别 / 90
Lisp-1 与 Lisp-2 / 91

第3章 编程语言的新潮流

3.1 语言的设计......97

客户端与服务器端 / 97
向服务器端华丽转身 / 98
在服务器端获得成功的四大理由 / 99
客户端的 JavaScript / 100
性能显著提升 / 101
服务器端的 Ruby / 102
Ruby on Rails 带来的飞跃 / 102
服务器端的 Go / 103
静态与动态 / 104
动态运行模式 / 105
何谓类型 / 105
静态类型的优点 / 106
动态类型的优点 / 106
有鸭子样的就是鸭子 / 107
Structural Subtyping / 108
小结 / 108

3.2　Go 109
New（新的）/ 109
Experimental（实验性的）/ 109
Concurrent（并发的）/ 110
Garbage-collected（带垃圾回收的）/ 110
Systems（系统）/ 111
Go 的创造者们 / 111
Hello World / 112
Go 的控制结构 / 113
类型声明 / 116
无继承式面向对象 / 118
多值与多重赋值 / 120
并发编程 / 122
小结 / 124

3.3　Dart 126
为什么要推出 Dart？ / 126
Dart 的设计目标 / 129
代码示例 / 130
Dart 的特征 / 132
基于类的对象系统 / 132
非强制性静态类型 / 133
Dart 的未来 / 134

3.4　CoffeeScript 135
最普及的语言 / 135
被误解最多的语言 / 135
显著高速化的语言 / 136
对 JavaScript 的不满 / 138
CoffeeScript / 138
安装方法 / 139
声明和作用域 / 139
分号和代码块 / 141
省略记法 / 142
字符串 / 143
数组和循环 / 143
类 / 145
小结 / 146

3.5　Lua 148
示例程序 / 149
数据类型 / 149
函数 / 150
表 / 150
元表 / 151
方法调用的实现 / 153
基于原型编程 / 155
和 Ruby 的比较（语言篇）/ 157
嵌入式语言 Lua / 157
和 Ruby 的比较（实现篇）/ 158
嵌入式 Ruby / 159

第 4 章　云计算时代的编程

4.1　可扩展性..163
信息的尺度感 / 163
大量数据的查找 / 164
二分法查找 / 165
散列表 / 167
布隆过滤器 / 169
一台计算机的极限 / 170

DHT（分布式散列表）/ 171
Roma / 172
MapReduce / 173
小结 / 174

4.2　C10K 问题 175

何为 C10K 问题 / 175
C10K 问题所引发的“想当然” / 177
使用 epoll 功能 / 180
使用 libev 框架 / 181
使用 EventMachine / 183
小结 / 185

4.3　HashFold 186

HashFold 库的实现（Level 1）/ 187
运用多核的必要性 / 190
目前的 Ruby 实现所存在的问题 / 191
通过进程来实现 HashFold（Level 2）/ 191
抖动 / 193
运用进程池的 HashFold（Level 3）/ 194
小结 / 197

4.4　进程间通信 198

进程与线程 / 198
同一台计算机上的进程间通信 / 199
TCP / IP 协议 / 201
用 C 语言进行套接字编程 / 202
用 Ruby 进行套接字编程 / 204
Ruby 的套接字功能 / 205
用 Ruby 实现网络服务器 / 208
小结 / 209

4.5　Rack 与 Unicorn 210

Rack 中间件 / 211
应用程序服务器的问题 / 212
Unicorn 的架构 / 215
Unicorn 的解决方案 / 215
性能 / 219
策略 / 220
小结 / 221

第5章　支撑大数据的数据存储技术

5.1　键 – 值存储 225

Hash 类 / 225
DBM 类 / 226
数据库的 ACID 特性 / 226
CAP 原理 / 227
CAP 解决方案——BASE / 228
不能舍弃可用性 / 229
大规模环境下的键 – 值存储 / 230
访问键 – 值存储 / 230
键 – 值存储的节点处理 / 231
存储器 / 232
写入和读取 / 233
节点追加 / 233
故障应对 / 233
终止处理 / 235
其他机制 / 235
性能与应用实例 / 236
小结 / 236

5.2　NoSQL 237

RDB 的极限 / 237
NoSQL 数据库的解决方案 / 238
形形色色的 NoSQL 数据库 / 239
面向文档数据库 / 240
MongoDB 的安装 / 241
启动数据库服务器 / 243
MongoDB 的数据库结构 / 244
数据的插入和查询 / 244
用 JavaScript 进行查询 / 245
高级查询 / 246
数据的更新和删除 / 249
乐观并发控制 / 250

5.3　用 Ruby 来操作 MongoDB 251

使用 Ruby 驱动 / 251
对数据库进行操作 / 253
数据的插入 / 253
数据的查询 / 253
高级查询 / 254
find 方法的选项 / 256
原子操作 / 257
ActiveRecord / 259
OD Mapper / 260

5.4　SQL 数据库的反击 264

“云”的定义 / 264
SQL 数据库的极限 / 264
存储引擎 Spider / 265
SQL 数据库之父的反驳 / 265
SQL 数据库 VoltDB / 268
VoltDB 的架构 / 269
VoltDB 中的编程 / 270
Hello VoltDB! / 271
性能测试 / 273
小结 / 275

5.5　memcached 和它的伙伴们 276

用于高速访问的缓存 / 276
memcached / 277
示例程序 / 278
对 memcached 的不满 / 279
memcached 替代服务器 / 280
另一种键 – 值存储 Redis / 282
Redis 的数据类型 / 284
Redis 的命令与示例 / 285
小结 / 289

第 6 章　多核时代的编程

6.1　摩尔定律 293

呈几何级数增长 / 293
摩尔定律的内涵 / 294
摩尔定律的结果 / 295
摩尔定律所带来的可能性 / 296
为了提高性能 / 297
摩尔定律的极限 / 302
超越极限 / 303
不再有免费的午餐 / 304

6.2　UNIX 管道 305

管道编程 / 306
多核时代的管道 / 308
xargs——另一种运用核心的方式 / 309
注意瓶颈 / 311
阿姆达尔定律 / 311
多核编译 / 312
ccache / 313
distcc / 313
编译性能测试 / 314
小结 / 315

6.3　非阻塞 I / O 316

何为非阻塞 I / O / 316
使用 read(2) 的方法 / 317
边沿触发与电平触发 / 319
使用 read(2) + select 的方法 / 319
使用 read+O_NONBLOCK 标志 / 321
Ruby 的非阻塞 I / O / 322
使用 aio_read 的方法 / 323

6.4　node.js 330

减负 / 330
拖延 / 331
委派 / 332
非阻塞编程 / 333
node.js 框架 / 333
事件驱动编程 / 334
事件循环的利弊 / 335
node.js 编程 / 335
node.js 网络编程 / 337
node.js 回调风格 / 339
node.js 的优越性 / 340
EventMachine 与 Rev / 341

6.5　ZeroMQ..342

多 CPU 的必要性 / 342
阿姆达尔定律 / 343
多 CPU 的运用方法 / 343
进程间通信 / 345
管道 / 345
SysV IPC / 346
套接字 / 347
UNIX 套接字 / 349
ZeroMQ / 349
ZeroMQ 的连接模型 / 350
ZeroMQ 的安装 / 352
ZeroMQ 示例程序 / 352
小结 / 354

版权声明 356

第 1 章 编程的时间和空间

1.1 编程的本质

在一部古老的电影《星际迷航 4：抢救未来》①中有这样一个镜头：从 23 世纪的未来穿越时空来到现代（1986 年）的“进取号”乘务员，为了操作计算机（Classic Mac）而手持鼠标与“计算机”讲话。看来在星际迷航的世界中，用人类语言作为操作界面就可以指挥计算机工作了。

不过，现代的计算机还无法完全理解人类的语言。市面上也有一些可以用日语来操作的软件，但距离实用的程度还差得很远。计算机本来是为了运行由 0 和 1 组成的机器语言而设计的，但与此同时，对于人类来说，要理解这种二进制位所构成的序列到底代表什么意思，却是非常困难的。

因此，创造出一种人类和计算机都能够理解的语言（编程语言），并通过这样的语言将人类的意图传达给计算机，这样的行为就叫做编程。

话虽如此，但是将编程仅仅认为是“因为计算机无法理解人类语言才产生的替代品”，我觉得也是不合适的。人类的语言其实非常模糊，有时根本就不符合逻辑。

Time flies like an arrow.

这句话的意思是“光阴似箭”（时间像箭一样飞走了），不过 flies 也有“苍蝇”（复数形态）的意思，因此如果你非要解释成“时蝇喜箭”也未尝不可，只要你别去纠结“时蝇”到底是啥这种朴素的问题就好了。

另一方面，和自然语言（人类的语言）不同，编程语言在设计的时候就避免了模糊性，因此不会产生这样的歧义。使用编程语言，就可以将步骤更加严密地描述出来。

用编程语言将计算机需要执行的操作步骤详细描述出来，就成了软件。计算机的软件，无论是像文字处理工具和 Web 浏览器这样的大型软件，还是像操作系统这样的底层软件，全部都是用编程语言编写出来的。

① 电影原题为 *Star Trek IV: The Voyage Home*，是著名的《星际迷航》系列科幻电影的第 4 部作品，上映于 1986 年。（若无特殊说明，本书的脚注均为译者注）

编程的本质是思考

由于我几乎一整天都对着计算机,因此我的家人可能认为我的工作是和计算机打交道。然而,将编程这个行为理解成“向计算机传达要处理的内容”是片面的。这样的理解方式，和实际的状态并不完全一致。



图 1　编程不是和计算机打交道，而是和人打交道

的确，程序员都是对着计算机工作的，但作为其工作成果的软件（中的大部分）都是为了完成人类所要完成的工作而设计出来的（图 1)。因此，“人们到底想要什么？想要这些东西的本质又是什么？要实现这个目的严格来说需要怎样的操作步骤？”思考并解决这些问题，才是软件开发中最重要的工作。换句话说，编程的本质在于“思考”。

尽管看上去是和计算机打交道的工作，但实际上编程的对象还是人类，因此这是个非常“有人味”的工作。个人认为，编程是需要人来完成的工作，因此我不相信在将来计算机可以自己来编程。

我是从初三的时候开始接触编程的。当时父亲买了一台夏普的袖珍计算机（PC-1210)，可以使用 BASIC 来编程。虽然这台袖珍计算机只能输入 400 个步骤，但看到计算机可以按照我的命令来运行，仿佛自己什么都能做到，一种“万能感”便油然而生。

创造世界的乐趣

尽管已经过去了 20 多年，但我从编程活动中所感到的“心潮澎湃”却是有增无减。

这种心潮澎湃的感觉，是不是由创造新世界这一行为所产生的呢？我喜欢编程，多少年来从未厌倦，这其中最大的理由，就是因为我把编程看作是一项创造性的工作吧。

只要有了计算机这个工具，就可以从零开始创造出一个世界。在编程的世界中，基本上没有现实世界中重力和因果关系这样的制约，如此自由的创造活动，可以说是绝无仅有的。能够按照自己的意愿来创造世界，这正是编程的最大魅力所在（图 2)。

正如现实世界是由物理定律所支撑的一样，编程所创造的世界，是由程序员输入的代码所构筑的规则来支撑的。通过创造一个像 Ruby 这样的编程语言，我对此尤其感触颇深，不过，即

便只是编写一个很小的程序，其本质也是相同的。



图 2　编程的乐趣在于创造性

因此，正是因为具有创造性这样重要的特质，编程才吸引了包括我在内的无数程序员，投入其中而一发不可收拾。将来，如果真能够像在《星际迷航》的世界那样，只要通过跟计算机讲话就可以获取所有的信息，那么编程也许就变得没有那么必要了。

其实，在搜索引擎出现之后，类似的状况已经正在上演了。拿我的孩子们来说，他们也经常频繁地坐在电脑跟前，但却从来没有进行过编程。对他们来说，电脑只是一个获取信息的渠道，或者是一个和朋友交流的媒介而已。编程这种事，是“爸爸在做的一种很复杂的事”，他们觉得这件事跟自己没什么关系。

不过，通过编程来自由操作计算机，并创造自己的世界，这样的乐趣如果不让他们了解的话，我觉得也挺遗憾的。但这样的乐趣并不是通过强加的方式就能够感受到的，而且用强制的方式可能反而会在他们心里埋下厌恶的种子，对此我也感到进退两难。教育孩子还真是不容易呢。

编程所具有的创造性同时也有艺术的一面。在摄影出现之后，绘画已经基本上丧失了用于记录的功能，但即便如此，颇具艺术性的绘画作品还是层出不穷。将来，即便编程的必要性逐渐消失，可能我还是会为了艺术性和乐趣而继续编程的吧。其实，像《星际迷航》中的世界那样，“计算机，请给我打开一个 Debian GNU/Linux 8.0 模拟器，我要写个程序”，这样的世界也挺有意思的不是吗？

快速提高的性能改变了社会

我们来换一个视角。在计算机业界，有很多决定方向性问题的重要“定律”，其中最重要的莫过于“摩尔定律”了。摩尔定律是由美国英特尔公司创始人之一的高顿 · 摩尔[①]于 40 多年前的 1965 年，在其发表的论文中提出的，这个定律的内容如下：

① 高顿 · E · 摩尔（Gordon Earle Moore，1929—　），1968 年参与创立英特尔公司，此后曾担任过英特尔公司总裁、首席执行官、董事长等职位，现已退休。

LSI 中的晶体管数量每 18 个月增加一倍[①]。

LSI 的集成度每 18 个月就翻一倍，这意味着 3 年就可以达到原来的 4 倍，6 年就可以达到 16 倍，呈指数增长。因此，30 年后，我们来算算看，就可以达到原来的 100 万倍呢。LSI 的集成度基本上与 CPU 性能和内存容量直接相关，可以说，在这 40 年中，计算机的性能就是以指数关系飞速增长的。此外，集成度也可以影响价格，因此性能所对应的价格则是反过来呈指数下降的。

想想看，现在你家附近电子商店中售价 10 万日元左右（约合人民币 8000 元）的笔记本电脑，性能恐怕已经超过 20 多年前的超级计算机了（图 3）。况且，超级计算机光一个月的租金就要差不多 1 亿日元（约合人民币 800 万元），就连租都已经这么贵了，如果真要买下的话得花多少钱啊……

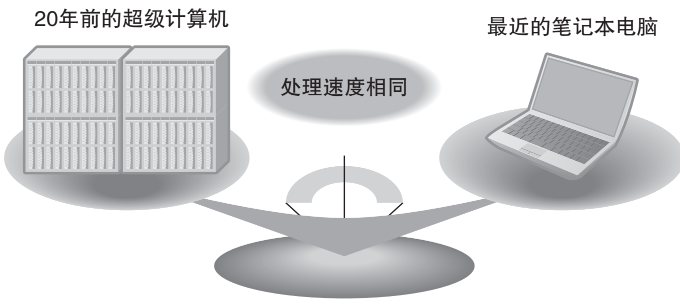


图 3　20 年前的超级计算机和现在的笔记本电脑性能处于同一水平

我在大学毕业之后就职的第一家公司里，用过一台索尼生产的 Unix 工作站，配置大概是这样的：

- CPU：摩托罗拉 68020 25MHz
- 操作系统：NEWS-OS 3.3a（基于 4.3BSD）
- 内存：8MB
- 硬盘：240MB
- 价格：定价 155 万日元（约合人民币 12 万元）

① LSI 指“大规模集成电路”。摩尔定律源于高顿 • 摩尔于 1965 年 4 月 19 日发表在《电子学》杂志第 114 页的一篇题为《让集成电路填满更多的组件》的文章，原文中的说法是“每年增加一倍”。1975 年，摩尔将这一定律修改为“每两年增加一倍”。摩尔曾公开表示，现在普遍流行的“每 18 个月增加一倍”的说法，并非由他本人提出，而是出自于英特尔公司的一位名叫 David House 的同事。

现在我几乎不敢相信索尼曾经生产过 UNIX 工作站，以至于连“工作站”（Workstation）这个词本身都已经几乎被淘汰了。工作站曾经指的是那些工程上使用的、性能比一般个人电脑要高一些的计算机（大多数情况下安装的是 UNIX 系操作系统）。

这台工作站也曾经是我最初开始编写 Ruby 所使用的机器。现在我自己家里的计算机已经拥有 Core2 duo 2.4GHz 的 CPU、4GB 内存和 320GB 硬盘[①]，单纯比较一下的话，CPU 频率大约是那台工作站的 100 倍，内存容量大约是 500 倍，硬盘容量大约是 1300 倍。这两台计算机的发售时间大约差了 18 年，按照摩尔定律来计算，集成度的增加率应该为 64 倍，可见内存和硬盘容量的增加速度已经远远超过摩尔定律所规定的速率了。

在当时的网络上，电子邮件和网络新闻组[②]是主流，网络通信还是在电话线路上通过调制解调器（Modem）来进行的。回头翻翻当时的杂志，看到像“9800bit/s 超高速调制解调器售价 198 000 日元（约合人民币 1.6 万元）”这样的广告还是感到挺震惊的。最近我们已经很少见到模拟方式的调制解调器了，我最后见过的调制解调器速度为 56Kbit/s，售价大约数千日元。

这正是摩尔定律的力量。在这个业界的各个领域中都经历着飞跃式的成长，近半个世纪以来，与计算机相关的所有部件，都随着时间变得性能更高、容量更大、价格更便宜。

在摩尔定律的影响下，我们的社会也发生了翻天覆地的变化。计算机现在已经变得随处可见，这应该说是摩尔定律为社会所带来的最大变化了吧。

我现在用的手机是 iPhone，这个东西与其说是个手机，不如说是一个拥有通信功能的迷你计算机。作为玩具它实在是很有趣，但因为整天鼓捣它还是被家里人给了差评。这样一个东西花几万日元就能买到，不得不感叹文明的进步。差不多在同样的时间，我给我的一个女儿买了一部普通的手机，这部手机跟 iPhone 不一样，只是那种一般的多功能机[③]，但仔细一看，这种手机也能上网，还装有 Web 浏览器、电子邮件、日程表等软件，也算得上一台不错的计算机了。

当初，让我感到最惊奇的是这个手机上居然安装了 Java 虚拟机，这样一来说不定能运行 JRuby 呢。不光是日本，全世界的人现在都能拥有这样的便携式计算机，并通过无线网络联系在一起，这样的情景在 20 年前简直是很难想象的。因此可以说，计算机的大规模普及，甚至改变了整个社会的形态。

---

① 这是 2009 年当时的配置，2012 年现在的配置为 Core i7 2.7GHz（双核 + 超线程），8GB 内存、300GB 固态硬盘（SSD）。经过 3 年的时间，好像并没有太大的进步呢。（原书注）

② 网络新闻组这个词现在也已经淘汰了，简单来说，它是一种类似分布式网络讨论组的东西。（原书注）

③ 原文是“ガラケー”，即“ガラパゴス・ケータイ”的简称，这是一个日本独有的说法。“ガラパゴス”指的是加拉帕戈斯群岛，位于太平洋东部，由于与世隔绝，岛上的生态系统十分独特，栖息着巨蜥、巨龟等其他地方没有的奇特动物。这个词暗示在智能手机快速普及的今天，日本市场中还存在着的很多仿佛与世隔绝般和主流技术格格不入的功能和服务，以及带有这些功能和服务的手机，这也算是日本手机市场的一大特色。

以不变应万变

由摩尔定律所引发的计算机方面的变化可以用翻天覆地来形容，但也并不是所有的一切都在发生变化（图 4）。

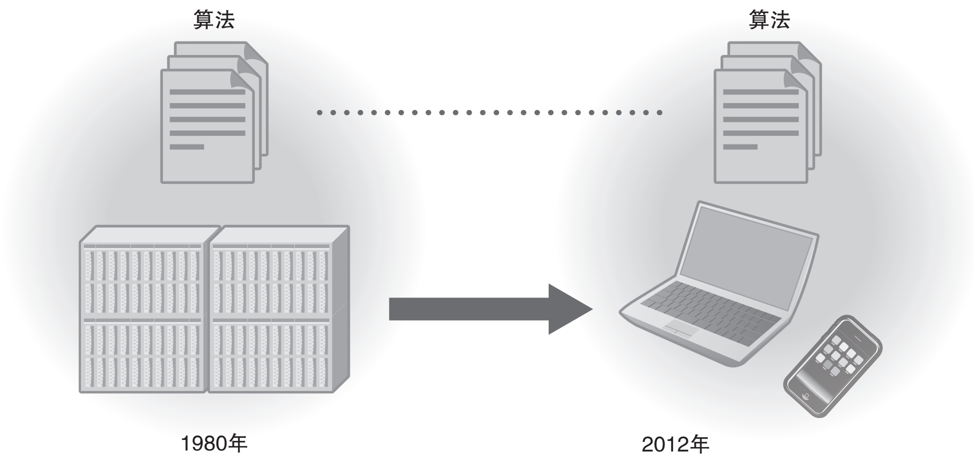


图 4　计算机在不断进化，而算法则保持不变

比如说，算法就能以不变应万变。被称为最古老算法的辗转相除法[①]，是在公元前 300 年左右被提出的。此外，计算机科学中的大多数基本算法都是在 20 世纪 60 年代被提出的。

我们来想想看电子邮件的情形。15 年前，几乎没什么人会使用电子邮件，但现在，电子邮件成了大家身边如影随形的工具，甚至有不少人一天到晚都在拿手机收发邮件。邮件影响了很多人的生活，甚至改变了我们的生活方式。

然而，邮件的基础技术却是出人意料地古老。世界上第一封电子邮件是在 1971 年发送的，而现在包括手机邮件在内所遵循的 RFC822 规范则是在 1982 年制定的，差不多是距今 30 多年前的东西了。此外，现在依然作为主流而被广泛使用的 TCP/IP 互联网通信协议也差不多是在那个时候制定的。

也就是说，这个技术本身是很早以前就存在的，只是一般人不知道而已。而更重要的一个原因是，人类自身的变化并没有那么快。读一读《圣经》之类的古典著作你就会惊奇地发现，人类从几千年前到现在所纠结的那些事情几乎没什么变化。从人类的本质来看，技术的进步只

① 辗转相除法，又称欧几里得算法，是欧几里得《几何原本》中记载的一种求两个自然数最大公约数的算法，是目前为止已知的最古老的算法。用现代算法的概念来说，它是一种递归算法。

不过是些细枝末节的改变罢了。

摩尔定律所带来的变化，并不是改变了人类自身以及计算的本质，而是将以往非常昂贵的计算机，以及只有特殊部门才需要的东西，普及到“老百姓”的手上。从这个侧面来讲，它所带来的变化的确是十分巨大的。

摩尔定律的局限

无论如何，在这 40 年里，摩尔定律的确在一直改变着世界，但是这个定律真的是完美的吗？

呈指数增长的趋势在如此长的时期内能够一直成立，这本身就很不自然。实际上，这个看似无敌的摩尔定律，最近也仿佛开始显露出一些破绽。我们可以预料到，在不远的将来，一定会出现一些因素，对摩尔定律的继续生效构成障碍。

首先是物理定律的局限。LSI 也是现实世界中物理存在的东西，自然受到物理定律的制约。在这 40 年里，LSI 一直在不断变得更加精密，甚至快要到达量子力学所管辖的地盘了①。当 LSI 的精密化达到这种程度，日常生活中一些从来不必在意的小事，都会变成十分严重的问题。

第一个重要的问题是光速。光速约为每秒 30 万千米，即 1 秒钟可以绕地球 7 圈半，这个数字十分有名，连小孩子都知道，不过正是因为光速实在太快，在日常生活中我们往往可以认为光速是无穷大的。

然而，CPU 的时钟频率已经到达了 GHz 尺度，比如说，在 3GHz 的频率下，波形由开到关（即 1 个时钟周期）的时间内，光只能前进 10cm 的距离。

而且，最近的 LSI 中电路的宽度已经缩小到只有数十纳米（nm），而 1nm 等于 100 万分之一 mm，是一个非常小的尺度，在 1nm 的长度上，只能排列几个原子，因此像这样在原子尺度上来制造电路是相当困难的。

LSI 中的电路是采用一种印刷技术②印上去的，在这样细微的尺度中，光的波长甚至都成了大问题，因为如果图像的尺寸比光的波长还小，就无法清晰地转印。可见光的波长范围约为 400 ~ 800nm，因此最近 45nm 制程的 LSI 是无法用可见光来制造的。

① 经典力学和经典电动力学都是用来描述宏观系统的，当集成电路精密程度达到原子级别的微观系统时，经典物理学就显得力不从心了，这就必须依靠用来描述微观物质的量子力学。

② 这里指的是“光刻法”（Photolithography）。

在这种原子尺度的电路中，保持绝缘也是相当困难的。简单来说，就是电流通过了原本不该通过的地方，这被称为漏电流。漏电流不但会浪费电力，某些情况下还会降低 LSI 的性能。

漏电流还会引发其他的问题，比如发热。随着 LSI 越来越精密，其密度也越来越高，热密度也随之提高。像现在的 CPU 这样高密度的 LSI，其热密度已经跟电熨斗或者烧烤盘差不多高了，因此必须用风扇等装置持续进行降温。照这个趋势发展下去，热密度早晚要媲美火箭的喷气口，如果没有充分的散热措施，连 LSI 本身都会被熔化。

由于漏电流和热密度等问题，最近几年，CPU 的性能提高似乎遇到了瓶颈。大家可能也都注意到了，前几年在店里卖的电脑还都配备了 3GHz、4GHz 的 CPU，而最近主流的电脑配置却是清一色的 2GHz 上下。造成这个现象的原因之一就是上面提到的那些问题，使得 CPU 一味追求频率的时代走到了尽头。此外，现在的 CPU 性能对于运行 Web 浏览器、收发邮件等日常应用已经足够了，这也是一个原因。

看了上面这些，大家可能会感到称霸了 40 多年的摩尔定律就快要不行了，不过英特尔公司的人依然主张“摩尔定律至少还能维持 10 年”。实际上，人们可以使用特殊材料来制造 LSI，以及使用 X 光代替可见光来进行光刻的转印等，通过这些技术的手段，摩尔定律应该还能再维持一阵子。

此外，由于通过提高单一 CPU 的密度来实现性能的提升已经非常困难，因此在一个 LSI 中集成多个 CPU 的方法逐渐成为主流。像英特尔公司的 Core2 i5、i7 这样在一个 LSI 上集成 2 ~ 8 个 CPU 核心的“多核”（Multi-core）CPU，目前已经用在了普通的电脑中，这也反映了上面提到的这一趋势。

比起拥有复杂电路设计的 CPU 来说，内存等部件由于结构简单而平均，因此其工艺的精密化更加容易。今后一段时间内，CPU 本身的性能提升已经十分有限，而多 CPU 化、内存容量的增大、由硬盘向半导体 SSD 转变等则会成为主流。

社会变化与编程

前面我们讨论了摩尔定律和它所带来的变化，以及对今后趋势的简单预测。多亏了摩尔定律，我们现在才可以买到大量高性能低价格的计算机产品。那么这种变化又会对编程产生怎样的影响呢？

我最早接触编程是在 20 世纪 80 年代初，在那个时候，使用电脑的目的就是编写 BASIC 程序。无论是性能还是容量，那个时候的计算机都非常差劲，根本无法与现在的计算机相提并论，此

外，还必须使用 BASIC 这种十分差劲的编程语言，这种环境对于编程的制约是相当大的。当时，我编写了许多现在看起来很不起眼的游戏，还对差劲的 BASIC 和计算机性能感到十分不爽，一边立志总有一天“一定要用上正经的计算机”，一边搜集着书本杂志中的信息做着自己的“春秋大梦”。

而另一方面，现在计算机已经随处可见，拿着手机这样的个人计算设备的人也不在少数。我的孩子们所就读的学校里，设有与理科教室、音乐教室等并列的电脑教室，有时也会用计算机来进行授课。这样一个时代中的年轻人，他们对于编程这件事又怎么看呢？

由于职业的关系，我家里有很多台计算机，算上不怎么经常用的，可以说计算机的数量比家里人的数量还要多①，当然，要是再算上手机之类的话，那就更多了。即便是生活在这样充满计算机的家庭中，孩子们对于编程貌似也没有什么兴趣。

那么，他们用计算机都做些什么事呢？比如用邮件和博客与朋友交流，用维基百科查阅学习上所需要的信息，还有在 YouTube 上看看动画片之类的。

上初中时学校曾经组织过用一种叫做“Dolittle”的编程语言②来做实习，孩子们也好像也挺感兴趣，不过并没有再进一步发展为真正的编程。对于他们来说，上上网站、看看 YouTube、发发邮件，有时候玩玩网购和在线竞拍，这些已经足够了。

我一个学生时代的朋友，现在正在大学任教，他对我说，现在信息技术类专业不但不如以前热门，而且招进来的学生中有编程经验的比例也下降了。这似乎意味着，计算机的普及率提高了，但是编程的普及率却一点都没有提高，真是令人嗟叹不已。

我猜想，大概是由于随着软件的发展，不用编程也可以用好计算机，因此学习编程的动力也就没有那么强了。此外，现在大家都认为软件开发是一份非常辛苦的工作，这可能也是导致信息技术类专业人气下滑的一个原因。

话虽如此，但并不是说真的一点希望都没有了。这几年来，我在一个叫做“U20 Pro Con”的以 20 岁以下青少年为对象的编程大赛中担任评委，每年的参赛作品中，总能见到一些水平非常高的程序。

也许是因为我担任评委的缘故，每年当我看到有自制编程语言方面的参赛作品时，总会感到十分震惊和欣慰。在我自己还是高中生的时候，虽然也想过创造一种编程语言，但完全不知道该怎样去做，到头来毫无进展。从这个角度来看，这些参赛的年轻人能够完整设计并实现一

① 松本行弘有 4 个孩子，因此家里一共有 6 口人。

② Dolittle 是由大阪电气通信大学的兼宗进等人开发的一种教育用编程语言，主页：http://dolittle.eplang.jp/。

种编程语言，比当年的我可优秀多了，因此我对他们将来的发展充满期待。

在这个世界上也有一些人，即便不去培养，他们也拥有想要编程的欲望，这样的人虽然只是小众，但他们会通过互联网获取丰富的知识，并不断攀登编程领域的高峰。编程的领地不会像计算机的普及那样飞速地扩展，但水平最高的人，水平却往往变得越来越高。这样的状况是我们希望看到的呢，还是不希望看到的呢？我也没办法做出判断。

现代社会已经离不开计算机和驱动计算机的软件了，从这个角度来说，我希望有更多的人能够积极地参与到编程工作中来。此外，我也希望大家不仅仅是将软件开发作为一份工作来做，而是希望更多的人能够感受到软件开发所带来的那种“创造的乐趣”和“心潮澎湃的感觉”。

1.2 未来预测

没有哪个人能够真的看到未来，也许正是因为如此，人们才想要预知未来，并对预言、占卜等方式充满兴趣。以血型、出生日期、天干地支、风水等为依据的占卜非常热门，事实上，杂志和早间电视节目中每次都有占卜的内容。

这些毫无科学依据的占卜方式是不靠谱的，虽说如此，占卜却还是大肆流行起来，其中有这样一些理由。

首先，最大的理由莫过于"巴纳姆效应"[①]了。"巴纳姆效应"是一种心理学现象，指的是将一些原本是放之四海而皆准的、模棱两可的一般性描述往自己身上套，并认为这些描述对自己是准确的。比如，找一些受试者做一份心理测试问卷，无论受试者如何回答问卷上的问题，都向他们提供事先准备好的内容差不多的测试结果，大多数的受试者都会认为这个结果对自己的描述非常准确。你觉得"占卜好准啊"，其实多半都是巴纳姆效应所导致的。即使是随便说说的一些话，也会有人深信不疑，这说不定是人类的一种本能吧。人类的心理到底为什么会拥有这样的性质呢？

其次，有很多算命先生和自称预言家的人，实际上都是利用了被称为"冷读术"（Cold reading）和"热读术"（Hot reading）的技巧，来让人们相信他们真有不同寻常的"能力"。

冷读术，就是通过观察对方言行举止中的一些细微之处来进行揣测的技巧，就像夏洛克·福尔摩斯对他的委托人所运用的那种技巧差不多。例如通过说话的口音来判断出生地，通过衣服上粘着的泥土来判断对方之前去过什么地方等等。冷读术中的"冷"代表"没有事先准备"的意思。

相对地，热读术则是通过事先对对方进行详细的调查，来准确说出对方的情况（逢场作戏）。通过事先调查，掌握对方的家庭构成、目前所遇到的问题等等，当然能够一语中的，再加上表演得像是拥有超能力一样，总会有人深信不疑的。

① 实际上，巴纳姆效应（Barnum effect）的发现者是心理学家伯特伦·福勒（Bertram Forer，1914—2000），因此也被称为福勒效应。而巴纳姆（P.T. Barnum）是一位著名的美国马戏业者，之所以用他的名字命名，是因为他曾经说过，"我的表演之所以受欢迎，是因为其中包含了每个人都喜欢的成分"，因此"每一分钟都会有人上当受骗"。

结论，占卜之类的方法都不靠谱，它们都是不科学的。那么，有没有科学一点的方法能够预测未来呢？比如说，像艾萨克·阿西莫夫的基地系列中所描写的心理史学那样。[①]

科学的未来预测

心理史学是阿西莫夫所创造的虚构学科。用气体分子运动论来类比，我们虽然无法确定每个气体分子的运动方式，但对于由无数气体分子所组成的气体，我们却可以计算出其整体的运动方式。同样地，对于由无数的人所组成的集团，其行为也可以通过数学的方法来进行预测。这样一类比的话，是不是感到很有说服力呢？

基地系列正是以基于心理史学的未来预测为轴，描写了以整个银河系为舞台，数兆人类的数千年历史。

然而，在现实中，特定个人的行动往往能够大幅左右历史的走向，即便是从整体来看，用数学公式来描述人类的行为还是太过复杂了，心理史学也许只能停留在幻想中而已。虽然心理史学只是一门完全虚构的学科，但这并不意味着不可能通过科学的方法来预测未来。虽然我们无法对未来作出完全准确的预测，但在限定条件下，还是可以在一定概率上对未来作出预测的，尤其是当我们要预测的不是未来人类的行动，而是纯粹预测技术发展的情况下。因此，IT 领域可以说是比较容易通过上述方式进行未来预测的一个领域了吧。

IT 未来预测

之所以说 IT 领域的未来比较容易预测，最大的一个理由是：从计算机的出现到现在已经过了约半个世纪，但在这 40 多年的时间里，计算机的基本架构并没有发生变化。现在主流的 CPU 架构是英特尔的 x86 架构，它的基础却可以追溯到 1974 年问世的 8080，而其他计算机的架构，其根本部分都是大同小异。这意味着计算机进步的方向不会有什么很大的变化，我们有理由预测，未来应该位于从过去到现在这个方向的延长线上（图 1）。

如果像量子计算机这样和现在的计算机架构完全不同的东西成为主流的话，我们的预测也就不成立了，不过还好，在短时间内（比如 5 年之类的）这样的技术应该还无法实现。此外，

① 艾萨克·阿西莫夫（Isaac Asimov，1920—1992）是著名的美国科幻小说家，他的作品中提出了很多对现代科幻小说甚至是现代科学技术影响深远的概念，如著名的“机器人三定律”。基地系列（The Foundation Series）是阿西莫夫撰写的系列科幻小说作品，整部基地系列作品包括《基地三部曲》《基地后传》和《基地前传》三大部分。

在这个行业中，5 年、10 年以后的未来已经算是相当遥远了，即便预测了也没有什么意义。总之目前来看，这样的趋势还是问题不大的。

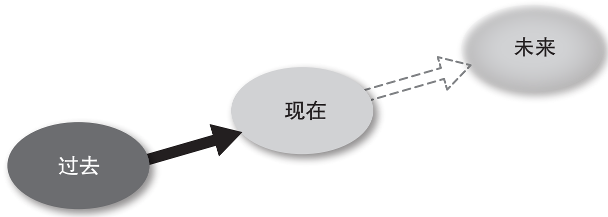


图 1　从过去到未来的发展方向

支配计算机世界"从过去到未来变化方向"的一个代表性理论,就是在 1-1 中已经讲解过的"摩尔定律"。

LSI 中的晶体管数量每 18 个月增加一倍。

在摩尔定律的影响下，电路变得更加精密，LSI 的成本不断降低，性能不断提高。其结果是，在过去的近 40 年中：

- ❑ 价格下降
- ❑ 性能提高
- ❑ 容量增大
- ❑ 带宽增加

这些都是呈指数关系发展的。呈指数关系，就像"一传十、十传百"一样，其增大的速度是十分惊人的（图 2）。

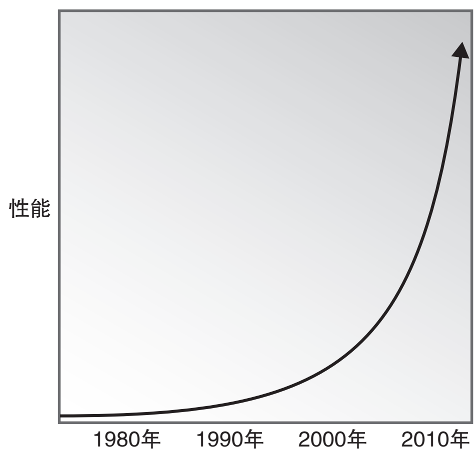


图 2　性能呈指数关系增加

而这一呈指数关系发展的趋势，预计在今后也会保持差不多的速度，这就是 IT 未来预测的基础。

另外一个需要考虑的问题，就是不同领域各自的发展速度。IT 相关的各种数值的确都在以指数关系增加，但大家的步调也并不是完全一致的。例如，相比 CPU 处理速度的提高来说，存储器容量的增加速度更快，而与上面两者相比，数据传输速度的增加又显得跟不上了。这种发展的不平衡也会左右我们的未来。

极限未来预测

下面我们来介绍一种预测未来的时候所用到的，名叫“极限思考法”的简单技巧。

曾提出过“极限编程”（eXtreme Programming，简称 XP）手法的肯特·贝克，在其著作《解析极限编程》中这样写道：①

> 当我第一次构建出 XP 时，我想到了控制板上的旋钮。每个旋钮都是一种（经验告诉我的）有效的实践，我将所有的旋钮都调到 10，然后等着看会出现什么情况。②

我们也可以用同样的方法来对未来作出预测。比如说，“如果计算机的价格越来越便宜，那当它便宜到极致的时候会怎么样呢？”“如果我们能够买到超高性能的计算机会怎么样呢？”“如果计算机的存储容量增大到超乎想象的程度会怎么样呢？”“如果网络带宽变得非常大的话会怎么样呢？”

大家怎么认为呢？

从价格看未来

首先，我们来看看价格。如果今后计算机的价格不断下降，这将意味着什么呢？我想这意味着两件事。第一，普通人所能拥有的计算机的性能将比现在大大提高；第二，现在还没有使用计算机的地方，以后都会安装上计算机。

这里有一个很有意思的现象，根据摩尔定律，关于计算机的很多指标都在发生剧烈的变化，但 PC 的价格似乎变化并没有那么大。1979 年发售的 NEC PC-8001 的定价为 16 万 8000 日元（约合人民币 1.3 万元），而现在主力 PC 的价格也差不多是在 10 万日元（约合人民币 8000 元）上下，即便考虑物价变化的因素，也还是出人意料地稳定。这可能意味着人类对于 PC 的购买力也就差不多只有这个程度，在不断提高的性能和价格之间在寻求一种平衡，因此我估计普及型计算机的价格今后也不太可能会大幅下降。关于将来 PC（PC 型计算机）的样子，我们会在“性能”一节中进行讨论。

① 肯特 • 贝克（Kent Beck，1961—　），是一位美国著名的软件工程师，极限编程思想的创始人之一。《解析极限编程——拥抱变化》（*Extreme Programming Explained: Embrace Change*），第一版出版于 1999 年，并于 2005 年出版了第二版。

② 这段译文引自中译本《解析极限编程——拥抱变化》，唐东铭译，人民邮电出版社（2002 年）。

关于在目前尚未开发的领域中安装计算机这件事，其实现在已经在上演了。例如，以前纯粹由电子电路所构成的电视机，现在也安装了 CPU、内存、硬盘等部件，从硬件上看和 PC 没什么两样，并且还安装了 Linux 这样的操作系统。此外，以前用单片机来实现的部分，现在也开始用上了带有操作系统的“计算机”，在这样的嵌入式系统中，软件所占的比例越来越大。今后，可以说外观长得不像计算机的计算机会越来越多，为这样的计算机进行软件开发的重要性也就越来越高。例如，现在由于内存容量和 CPU 性能的限制而无法实现的开发工具和语言，以后在“嵌入式软件”开发中也将逐渐成为可能。

从性能看未来

从近 10 年计算机性能变化的趋势来看，CPU 自身的性能提高似乎已经快要到达极限了。近几年，很多人会感觉到 PC 的时钟频率似乎到了 2GHz 就再也上不去了。这种性能提高的停滞现象，是由耗电、漏电流、热密度等诸多原因所导致的，因此从单一 CPU 的角度来看，恐怕无法再继续过去那样呈指数增长的势头了。

那么这样下去结果会怎样呢？要推测未来计算机的性能，最好的办法是看看现在的超级计算机。因为在超级计算机中为了实现高性能而采用的那些技术，其中一部分会根据摩尔定律变得越来越便宜，在 5 到 10 年后的将来，这些技术就会被用在主流 PC 中。

那么，作为现在超级计算机的代表，我们来看看 2012 年目前世界最快的超级计算机“京”[①]的性能数据（表 1）。虽然它的性能看起来都是些天文数字，但再过 20 年，这种程度的性能很可能就只能算是“一般般”了。

表1　超级计算机“京”的指标

性能	10000TFLOPS
价格	1120亿日元（约合人民币90亿元）
CPU数量	88128个
核心数量	705024个
内存	2.8PB（平均每个CPU拥有32GB）

说不定在不久的将来，1024 核的笔记本电脑就已经是一般配置了。如果是服务器环境的话，也许像现在的超级计算机这样数万 CPU、数十万核心的配置也已经非常普遍了。难以置信吧？

在这样的环境下，编程又会变成什么样子呢？为了充分利用这么多的 CPU，软件及其开发环境又会如何进化呢？

① 京（K computer）是设在日本理化学研究所的一台超级计算机，从 2005 年开始研发，2012 年 6 月正式宣告完成。其名称“京”表示它的运算速度可以达到每秒 1 京次（10000 万亿次，即 10^{16} 次）的级别。根据世界超级计算机 TOP500 排行榜的数据，在本书出版的当时（2012 年 4 月），“京”依然保持着世界最快超级计算机的纪录，但在 2012 年 6 月，这一纪录被美国 IBM 的超级计算机“红杉”（Sequoia）打破。

考虑到这样的环境，我认为“未来的编程语言”之间，应该在如何充分利用 CPU 资源这个方面进行争夺。即便是现在，也已经有很多语言提供了并行处理的功能，而今后并行处理则会变得愈发重要。如果能将多个核心的性能充分利用起来，说不定每个单独核心的性能就变得没有那么重要的。

从容量看未来

存储器的容量，即内存容量和外存（硬盘等）容量，是增长速度最快的指标。2012 年春，一般的笔记本电脑也配备了 4GB 的内存和 500GB 左右的硬盘，再加上外置硬盘的话，购买 2 ~ 3TB 的存储容量也不会花上太多的钱。一个普通人所拥有的存储容量能达到 TB 级，这在 10 年前还是很难想象的事情，而仅仅过了没多少时间，我们就可以在电子商店里轻松买到 TB 级容量的硬盘了。

那么，存储器容量的增加，会对将来带来哪些变化呢？大家都会想到的一点是，到底从哪里才能搞到那么多的数据，来填满如此巨大的容量呢？

实际上，这一点根本用不着担心。我们来回想一下，无论存储容量变得多大，不知怎么回事好像没过多久就又满了。为了配合不断增加的存储容量，图片数据和视频数据都变得更加精细，尺寸也就变得更大。另外，软件也变得越来越臃肿，占用的内存也越来越多。以前的软件到底是怎样在那么小的内存下运行得如此流畅的呢？真是想不通啊。

因此，问题是我们要如何利用这些数据呢？也许面向个人的数据仓库之类的数据分析工具会开始受到关注。当然，这种工具到底应该在客户端运行，还是在服务器端运行，取决于性能和带宽之间的平衡。

在存储器容量方面，与未来预测相关并值得关注的一个动向，就是访问速度。虽然容量在以惊人的速度增长，但读取数据的速度却没有按照匹配的速度来提高。硬盘的寻址速度没什么长进，总线的传输速度也是半斤八两。不过，像闪存这样比硬盘更快的外部存储设备，现在也已经变得越来越便宜了，由闪存构成的固态硬盘（Solid State Drive，SSD）已经相当普遍，完全可以作为硬盘的替代品。按照这个趋势发展下去，在不久的将来，说不定由超高速低容量的核心内置缓存、高速但断电会丢失数据的主内存（RAM），以及低速但可永久保存数据的外部存储器（HDD）所构成的层次结构将会消失，取而代之的可能将会是由大规模的缓存，以及高速且能永久保存数据的内存所构成的新的层次结构。如果高速的主内存能够永久保存数据，依赖过去结构的数据库等系统都将产生大规模的结构改革。实际上，以高速 SSD 为前提的数据库系统，目前已经在进行研发了。

从带宽看未来

带宽，也就是网络中数据传输的速度，也在不断增大。一般家庭的上网速度，已经从模拟调制解调器时代的不到 100Kbit/s，发展到 ADSL 时代的 10Mbit/s，再到现在光纤时代的超过 100Mbit/s，最近连理论上超过 1Gbit/s 的上网服务也开始面向一般家庭推出了。

网络带宽的增加，会对网络两端的平衡性产生影响。在网络速度很慢的时代，各种处理只能在本地来进行，然后将处理结果分批发给中央服务器，再由中央服务器进行统计，这样的手法十分常见。这就好像回到了计算机还没有普及,大家还用算盘和账本做着“本地处理”的时代。

然而后来，各种业务的处理中都开始使用计算机，每个人手上的数据都可以发送到中央计算机并进行实时处理。但由于那时的计算机还非常昂贵,因此只是在周围布置了一些被称为“终端”的机器，实际的处理还是由设在中央的大型计算机来完成的。那是一个中央集权的时代。

在那以后，随着计算机价格的下降，每个人都可以拥有自己的一台计算机了。由于计算机可以完成的工作也变多了，因此每个人手上的“客户端”计算机可以先完成一定程度的处理，然后仅仅将最终结果传送给位于中央的“服务器”，这样的系统结构开始普及起来，也就是所谓的“客户端 / 服务器系统”（Client-Server system），也有人将其简称为“CS 系统”。

然而，如果网速提高的话，让服务器一侧完成更多的处理，在系统构成上会更加容易。典型的例子就是万维网（World Wide Web，WWW）。在网速缓慢的年代，为了查询数据而去直接访问一个可能位于地球背面的服务器，这种事是难以想象的，如此浪费贵重的带宽资源，是要被骂得狗血淋头的。话说，现在的网络带宽已经像白菜一样便宜了，这样一来，客户端一侧只需要准备像“浏览器”这样一个通用终端，就可以使用全世界的各种服务了，如此美好的世界已经成为了现实。由于大部分处理是在服务器一侧执行的，因此乍看之下仿佛是中央集权时代的复辟，不同的是，现在我们可以使用的服务多种多样，而且它们位于全世界的各个角落。

但是，计算机性能和带宽之间的平衡所引发的拔河比赛并没有到此结束。近年来，为了提供更丰富的服务，更倾向于让 JavaScript 在浏览器上运行，这实际上是“客户端 / 服务器系统”换个马甲又复活了。此外，服务器一侧也从一台计算机，变成了由许多台计算机紧密连接所构成的云计算系统。换个角度来看的话，以前由一台大型机所提供的服务，现在变成由一个客户端 / 服务器结构来提供了。

今后，在性能和带宽寻求平衡的过程中，网络彼此两端的系统构成也会像钟摆一样摇个不停。从以往的情况来看，随着每次钟摆的来回，系统的规模、扩展性和自由度都能够得到提高，今后的发展也一定会遵循这样一个趋势。

小结

在这里，我们瞄准从过去到现在发展方向的延长线，运用极限思考法，尝试着对未来进行了预测。书籍是可以存放很久的，5 年、10 年之后再次翻开这本书的时候，到底这里的预测能不能言中呢？言中的话自然感到开心，没言中的话我们就一笑了之吧，胜败乃兵家常事嘛。

第 2 章

编程语言的过去、现在和未来

2.1 编程语言的世界

大家知道世界上最早的编程语言是什么吗？一般认为是 1954 年开始开发的 FORTRAN 语言。

然而，仔细想想看，到底什么才叫编程语言？如果将对机器的控制也看成是编写“程序”的话，那么编程的起源便可以追溯到杰卡德织机上面所使用的打孔纸带（图 1）。

1801 年，正值工业革命期间，杰卡德织机的发明使得提花编织的图案可以通过“程序”来自动完成。从前在各个家庭中也出现了自动纺织机，用于家庭作坊式的自动纺织生产，而杰卡德织机则相当于是这些家庭纺织机的放大版。我想那些自动纺织机应该也可以通过类似打孔纸带的东西来输入图案，当然，最近的年轻人恐怕都没有亲眼见过纺织机吧。

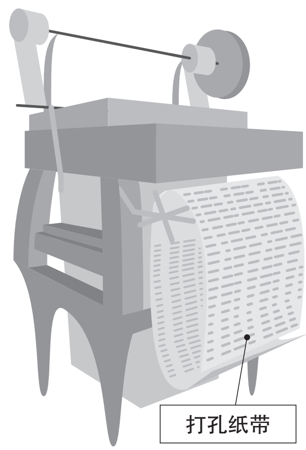


图 1　杰卡德织机的示意图

这种用打孔纸带来控制机器的想法，对各个领域都产生了影响。例如在英国从事通用计算机研发的查尔斯 · 巴贝奇，就在自制的“分析机”上用打孔纸带来输入控制程序。[①]遗憾的是，由于资金和其他一些问题，巴贝奇在生前未能将他的分析机制造出来。

不过，分析机的设计已经完成，用于分析机的程序也作为文档保留了下来。协助开发这些程序的，是英国诗人拜伦之女爱达 · 洛夫莱斯[②]，据说她和巴贝奇是师兄妹关系。如果不算分析机的设计者巴贝奇，那么世界上第一位程序员实际上是一位女性。为了纪念她，还有一种编程语言以她的名字 Ada 命名[③]。

说点题外话，在现在的编程界中，女性人数很少这一点是有目共睹的，尤其是在开源相关的活动上，男女比例达到 100 比 1 也不稀奇。其实，在计算机早期时代，有记录表明人们大都

① 查尔斯 • 巴贝奇（Charles Babbage，1791—1871），是英国数学家，计算机先驱，可编程计算机的发明者。分析机（Analytical Engine）是巴贝奇设计的一种机械式十进制通用计算机，虽然由于种种原因，它并未被真正制造出来，但其设计理念可以说是 100 年后电子通用计算机的先驱。

② 爱达 • 洛夫莱斯（Ada Lovelace，1815—1852），原名奥古斯塔 • 爱达 • 拜伦，洛夫莱斯是丈夫威廉 • 洛夫莱斯伯爵的姓氏。

③ 这里指的是 1980 年由美国军方设计的一种名为 Ada 的编程语言。

认为程序员应该是女性从事的工作，也许是人们将程序员和当时电话交换机的接线员（从业者中也是女性居多）看成是同一类型的工作吧。

在被称为世界上第一台计算机的 ENIAC[①]（1946 年）中，程序不是用打孔纸带，而是通过接电线的方式来输入的，我总觉得这是一种倒退。

不过，无论是接电线，还是打孔纸带，都不大可能实现复杂的程序，真正的程序恐怕还要等到存储程序式计算机出现之后。一般认为，世界上第一台存储程序式电子计算机，是 1949 年出现的 EDSAC[②]。

到了这个时候，所谓的“机器语言”就算正式问世了。当时的计算机程序都是用机器语言来编写的。那个时候不要说是编译器，连汇编器都还没发明出来呢，因此使用机器语言也是理所当然的事。

说到底，机器语言就是一串数字，将计算的步骤从指令表中查出对应的机器语言编码，再人工写成数列，这个工作可不容易。或者说，以前的人虽然没有意识到，但从我们现代人的角度来看，这种辛苦简直是难以置信。比如说，把引导程序的机器语言数列整个背下来，每次启动的时候手工输入进去；将机器语言指令表全部背下来，不用在纸上打草稿就能直接输入机器语言指令并正确运行——“古代”的程序员们留下了无数的光辉事迹（或者是传说），那时候的人们真是太伟大了。

然而有一天，有一个人忽然想到，查表这种工作本来应该是计算机最擅长的，那么让计算机自己来做不就好了吗？于是，人们用更加容易记忆的指令（助记符）来代替数值，并开发了一种能够自动生成机器语言的程序，这就是汇编器。

汇编器是用来解释“汇编语言”的程序，汇编语言中所使用的助记符，和计算机指令是一一对应的关系。早期的计算机主要还是用于数值计算，因此数学才是主宰。在数学的世界中，数百年来传承下来的“语言”就是算式，因此用接近算式的形式来编写计算机指令就显得相当方便。随后，FORTRAN 于 1954 年问世了。FORTRAN 这个名字的意思是：

```
FORmula TRANslator
  算式       翻译器
```

也就是说，编程语言是由编程者根据自己的需要发明出来的。早期的计算机，由于性能不足、

① ENIAC（Electronic Numerical Integrator And Computer，电子数值积分计算机），是由美国宾夕法尼亚大学穆尔电气工程学院设计建造的电子计算机，于 1946 年公布，为美国陆军的弹道研究实验室服务。

② EDSAC（Electronic Delay Storage Automatic Calculator，电子延迟存储自动计算器），是英国剑桥大学数学实验室的莫里斯 • 威尔克斯（Maurice Wilkes，1913—2010）教授及其团队于 1946 年开始设计建造的电子计算机，1949 年 5 月 6 日正式运行。

运算成本高，因此编写和维护程序都被看成是非人的工作，而编程语言正是其开始摆脱非人性的象征。

其实，由助记符自动生成机器语言的汇编器，以及由人类较易懂的算式型语句生成机器语言的编译器，当时都被认为是革新性的技术，被称为“自动编程”。此外，编译器开发技术的研究甚至被视为人工智能研究的一部分。

被历史埋没的先驱

一般大家都认为 ENIAC 是世界上第一台计算机，而 FORTRAN 是世界上第一个编程语言，然而，事实果真如此吗？我觉得有必要刨根问底一番。

实际上，如果仔细查阅一下计算机的历史，还是会发现一些不同观点的。

首先，世界上第一台计算机，其实应该是“阿塔纳索夫 – 贝瑞计算机”（Atanasoff-Berry Computer，简称 ABC），这台计算机的测试机完成于 1939 年，远比 ENIAC 要早。而且，ABC 在数值的表现方法上采用了现在广泛使用的二进制计算（ENIAC 为十进制计算），这也是 ABC 的其中一个先进之处。

ENIAC 甚至都不能算作是世界上第二台计算机。当时在第二次世界大战中与美国敌对的德国，开发出了一台用于土木工程计算的计算机 Z3[①]。这台计算机完成于 1941 年，和 ABC 一样，在数值表现上也采用了二进制。和由电子管组成的 ABC 和 ENIAC 不同，Z3 是继电器式计算机。遗憾的是，Z3 在 1944 年柏林轰炸中被毁。

那么，编程语言方面又如何呢？通过查阅资料发现，开发 Z3 的德国工程师康拉德 · 楚泽，于 1942 年至 1945 年间开发了一种名为 Plankalkül 的编程语言，比 FORTRAN 早了将近 10 年。[②] 然而，Plankalkül 只是被设计出来，而没有被正式发表，而且用于该编程语言的编译器也没有被开发出来。

Plankalkül 的设计直到 1972 年才被正式发表，而到第一个用于该语言的编译器正式实现，已经是 1998 年的事了。因此，如果论完整开发并能工作的编程语言，FORTRAN 作为最古老编程语言的地位还是无人能够撼动。

① 在 Z3 之前当然还开发了 Z1 和 Z2，但它们是用于特定用途的计算机，而并非通用计算机。（原书注）

② 康拉德 • 楚泽（Konrad Zuse，1910—1995），是一位德国土木工程师，计算机先驱。Plankalkül 被认为是世界上第一个非冯 • 诺依曼型高级编程语言，它的名称在德语中的意思是“用于规划的正式系统”（Formal system for planning）。

Plankalkül 由于种种原因被淹没在历史的长河中，因此它对后世的编程语言几乎没有产生影响，但是，它却考虑了如赋值语句、子程序、条件判断、循环、浮点小数计算、数组、拥有层次结构的结构体、断言、异常处理、目标搜寻等功能，其中一些甚至连 10 年后出现的 FORTRAN 都不具备，可见其先进性着实令人惊叹。

```
P1 max3（V0[:8.0],V1[:8.0],V2[:8.0]）=>
R0[:8.0]
max（V0[:8.0],V1[:8.0]）=> Z1[:8.0]
max（Z1[:8.0],V2[:8.0]）=> R0[:8.0]
END
P2 max（V0[:8.0],V1[:8.0]）=> R0[:8.0]
V0[:8.0] => Z1[:8.0]
（Z1[:8.0] < V1[:8.0]）-> V1[:8.0] => Z1[:8.0]
Z1[:8.0] => R0[:8.0]
END
```

图 2　用 Plankalkül 编写的程序

图 2 给出了一段 Plankalkül 程序，其中定义了用于对两个参数进行比较的子程序 max，以及利用这个子程序进而对三个参数进行比较的子程序 max3。其中所有的运算过程都被表示为“计算 => 结果保存位置”这样的形式，相当有意思。

编程语言的历史

在 FORTRAN 之后，20 世纪 50 年代末至 80 年代初，各种各样的编程语言如雨后春笋般相继出现，从而成就了一个编程语言研究飞速发展的鼎盛时期。

在这里，我们暂且抛开那些现在还健在的主流编程语言，而是将目光放在一些最近不怎么听说了，但却值得品味的编程语言上，并借此简单介绍一下编程语言的历史。

1. FORTRAN

话说回来，一开始我们还是先来讲讲 FORTRAN 这个主流语言吧。

FORTRAN 作为实质上的世界第一种编程语言，在数值计算领域建立了霸主地位。需要数值计算功能的物理学家等研究人员并不关心编程，对他们来说，计算速度有多快，是否能充分利用过去的成果，这两点才是最重要的。

从结果来看，FORTRAN 非常重视和过去的程序之间的兼容性，为此，它保留了一些现在已经几乎不再使用的编码风格，其中最典型的一个例子，就是 FORTRAN 的语法中所包含的下面这个难以置信的规则。

在程序中空格没有意义

这个规则的意思是，空格只是为了易读才加上去的，而编译器在编译程序时会把空格全部去掉。这样的语法，以现代编程语言的习惯来看简直是匪夷所思。为什么这样说呢？例如：

```
DO 100 I = 1,10
```

这样一行程序，它的语法和 DO 语句的语法是一致的，程序的意思是在循环中将从 1 到 10 的值依次赋值给变量 I，当循环结束之后跳转到行号为 100 的语句，到这里看起来也没有什么问题。

不过，如果我将 DO 语句中“1,10”中的逗号错打成句点的话，会发生什么事呢？也就是说：

```
DO 100 I = 1.10
```

我们之前已经说过，FORTRAN 会忽略所有的空格，那么这个语句就会被转换为：

```
DO100I=1.10
```

这样一来，由于这个语句不符合 DO 语句的语法，因此编译器会将这个语句理解为“将浮点小数 1.10 赋值给变量 DO100I”，编号为 100 的那一行则被彻底忽略掉了。也就是说，本来程序的意图是要执行一个循环，现在却变成了一个赋值语句，循环也就无效了。这可以说是在语法分析相关研究相对落后的年代设计出来的编程语言所特有的悲剧。

另一方面，追求计算速度的人们依然在使用着 FORTRAN，并且积累了一些十分大胆的优化技术。例如，为了最大限度利用超级计算机中装备的向量处理器，而自己改写 DO 循环以实现向量化等。

最近的超级计算机也几乎不再使用向量型结构，因此也很少会遇到必须使用 FORTRAN 的情况，使用超级计算机的研究人员更多地开始使用 Java 和 C++ 之类的语言。不过，即便如此，FORTRAN 也并没有要消亡的迹象。

此外，FORTRAN 本身也在不断进化，在最新版本中还实现了如编程抽象化、数据结构分配等功能，和其他先进编程语言相比毫不逊色。

2. COBOL

COBOL 这个名称是 COmmon Business Orientd Language（面向商业的通用语言）的缩写，相对于面向科学计算的 FORTRAN，COBOL 是作为面向商务计算而出现的编程语言，于 1959 年被开发出来。

基于下面的推理：

- ❑ 由于是面向商业计算，因此不需要太多的算式
- ❑ 对于从事商业活动的“一般大众”来说，采用更接近日常表达方式（英语）的语法，比采用数学算式更容易理解

COBOL 采用了类似英语的语法，但对于我这个日本人来说，把：

```
AB = A * B
```

写成：

```
MULTIPLY A BY B GIVING AB.
```

真的更容易理解吗？恐怕要打一个大大的问号。

不过，由于积累了大量过去的数据，COBOL 直到 21 世纪的今天依然健在。虽然在计算机类杂志中已经看不到关于 COBOL 的文章，但也有说法称，COBOL 依然是现在使用最广泛的一种编程语言。

实际上，在我供职的网络应用通信研究所中，有一半的技术人员是用 Ruby 来进行软件开发，剩下一半则是使用 COBOL。不过我们用的硬件只是普通的 PC 服务器，操作系统用的是 Linux，其他事务监视器等工具用的也都是开源软件，这倒是还挺少见的。

那么，COBOL 到底好不好用呢？从像我这样没用过 COBOL 的人的角度来看，一定会觉得“这啥啊，跟别的程序差别太大了吧”，不过 COBOL 的技术人员会主张说“如果要实现把账簿里的数字从右边移到左边，即便在现在 COBOL 也是效率最高的”。即使排除他们已经用惯了 COBOL 这个因素，这个评价还是相当值得回味的。

COBOL 也在不断进化，据说在最新版本中还增加了面向对象的功能。

3. Lisp

Lisp 和 FORTRAN、COBOL 一起并称为“古代编程语言三巨头”（我命名的），它的名字意思是：

```
LISt   Processor
表      处理器
```

Lisp 是一种特殊的编程语言，因为它原本并不是作为编程语言，而是作为一种数学计算模型来设计的。Lisp 的设计者约翰 · 麦卡锡设计了一种以 Lambda 演算为基础的“图灵完全”[①]的

① 约翰 • 麦卡锡（John McCarthy，1927—2011）是美国著名的计算机科学家，于 1971 年因人工智能方面的贡献获得图灵奖。Lambda 演算（Lambda calculus）是由阿隆佐 • 邱奇（Alonzo Church，1903—1995）提出的一种十分简洁的计算模型，是 Lisp 等函数型编程语言的理论基础。它不同于通用图灵机的计算模式，但它是图灵完全的。图灵完全（Turing complete）是指一种计算模型拥有和通用图灵机相同的计算能力，即一种计算模型可以实现对目前已知所有算法的描述。目前大部分通用编程语言（如 C、Java 等）都是图灵完全的。

计算模型，而 Lisp（的起源）则是为了描述这一模型而设计出来的。

不过，麦卡锡并没有想到将其作为一种计算机语言来使用，直到他实验室的一位研究生史帝芬·罗素用 IBM 704 机器语言实现了描述计算模型的万能函数 eval（求值函数），Lisp 才作为一种编程语言正式诞生。[①]

此外，Lisp 是早期能够对数值以外的值进行处理的语言。相对于 FORTRAN 和 COBOL 这样只能进行数值计算的语言来说，Lisp 擅长于表结构这样的非数值型处理，因此，Lisp 被广泛用于人工智能这样的领域中。

在 20 世纪 60 年代，一般人还不会为了计算以外的目的使用计算机，因此 Lisp 主要在大学和研究所等地方传播开来。

现在的计算机对字符串处理和通信等非数值处理已经成为主流，而 Lisp 则是这样一个时代的先驱者。此外，因 Java 而广为人知的垃圾回收和异常处理等机制，实际上最初是由 Lisp 发明的。因此，Lisp 对现在的计算机科学基础的构筑做出了极大的贡献，尽管使用者寥寥，但现在它依然是用于实用系统开发的现役编程语言。

4. SNOBOL

早期的 Lisp 所能够处理的数据只有表和符号，并不擅长处理字符串。而一种专门用于字符串处理的语言，在历史上扮演了重要的角色，它就是 SNOBOL（发音近似 Snow Ball）。SNOBOL 的名字意思是：

```
StriNg Oriented symBOlic Language
 字符串    面向    符号化   语言
```

其中 N 和 BO 取的位置好像很奇怪，不过别太在意就是了。SNOBOL 的革新性在于它是以模板（Pattern）为中心来进行字符串处理的，可以说是现在的 AWK、Perl 和 Ruby 等语言的祖先。不过 SNOBOL 的字符串模板并不是正则表达式，而是采用了类似 BNF（巴科斯 – 诺尔范式）[②]的写法。

由这样的编程语言所积累的经验，可以说是加快了计算机向数值计算以外的方向发展的步伐。

① IBM 704 是由 IBM 公司于 1954 年推出的内置浮点计算功能的大型计算机，FORTRAN 和 Lisp 等编程语言都是在 IBM 704 上开发出来的。

② 巴科斯 – 诺尔范式（Backus-Naur Form），是由约翰 • 巴科斯（John Backus，1924—2007）和彼得 • 诺尔（Peter Naur，1928—　）首先引入的用来描述计算机语言语法的符号集，是一种描述上下文无关文法的语言。

图 3 展示了一段用 SNOBOL 的模板匹配来解 N 皇后问题[①]的程序。当模板匹配时跳转到一个明确标明的标签，这样的风格现在已经不怎么见得到了。

```
* N queens problem, a string oriented version to
*   demonstrate the power of pattern matching.
* A numerically oriented version will run faster than this.
        N = 5
        NM1 = N - 1; NP1 = N + 1; NSZ = N * NP1; &STLIMIT = 10
 ** 9; &ANCHOR = 1
        DEFINE('SOLVE(B)I')
* This pattern tests if the first queen attacks any of the othe
rs:
        TEST = BREAK('Q') 'Q' (ARBNO(LEN(N) '-') LEN(N) 'Q'
+              | ARBNO(LEN(NP1) '-') LEN(NP1) 'Q'
+              | ARBNO(LEN(NM1) '-') LEN(NM1) 'Q')
        P = LEN(NM1) . X LEN(1); L = 'Q' DUPL('-',NM1) ' '
        SOLVE()        :(END)
SOLVE   EQ(SIZE(B),NSZ)             :S(PRINT)
* Add another row with a queen:
        B = L B
LOOP    I = LT(I,N) I + 1 :F(RETURN)
        B TEST :S(NEXT)
        SOLVE(B)
* Try queen in next square:
NEXT    B P = '-' X :(LOOP)
PRINT   SOLUTION = SOLUTION + 1
        OUTPUT = 'Solution number ' SOLUTION ' is:'
PRTLOOP B LEN(NP1) . OUTPUT = :S(PRTLOOP)F(RETURN)
```

图 3　用 SNOBOL 来解 N 皇后问题的程序

5. 数学性语言

刚才已经讲过，Lisp 是由数学而诞生的编程语言，不过这样的语言可不止 Lisp 一个。

例如，Prolog 就是一个以一阶谓词逻辑为基础的编程语言。20 世纪 80 年代，日本政府主导的第五代计算机计划[②]中就采用了 Prolog（准确地说应该是以 Prolog 为基础的逻辑型编程语言），使得其名声大噪，但当第五代计算机计划逐渐淡出人们的视线之后，Prolog 也随之销声匿迹了。不过，逻辑型编程语言所具备的“联合”（Unification）匹配模式，也在最近备受关注的函数型编程语言中被继承了下来。

① N 皇后问题（N-Queen Problem），最初为 8 皇后问题，即如何在国际象棋棋盘（8×8）上摆放 8 个皇后，使得它们彼此都无法直接吃掉对方（即彼此不能处于同一直线或斜线上）。8 皇后问题共有 12 个独立解（将旋转、对称解作为同一解的话），而后来便由此引申出 N 皇后问题（此时皇后个数为 N，且棋盘大小同时为 N×N）。

② 第五代计算机计划，是由日本通商产业省（现经济产业省）主导的一项国家开发计划，目的是开发“可高速执行谓词逻辑推理的并行逻辑计算机及其操作系统”。整个计划从 1982 年启动，共耗时 10 年，花费 570 亿日元，于 1992 年宣布结束，并称“已达到预期目标”。

以数学为基础的编程语言中，还有另外一个派系被称为函数型编程语言，例如 ML、Haskell 等就属于这一类。这一类数学性语言，比起机器的处理方式，更倾向于直接表达要表达的概念，也就是说相对于 How（如何实现），更倾向于通过 What（想要什么）来表达问题。这样可以不被机器的处理方式所左右，将问题抽象地表达出来，这是一种非常先进的特性。关于这一点，在后面“未来的编程语言”一节中还会进行详细介绍。

6. 主流语言

众人皆知的那些编程语言就没必要特地在这里介绍了吧。作为系统描述语言的 C 和 C++、作为面向商务的语言而如日中天的 Java，以及在 Web 领域中十分热门的 Ruby、Perl、Python、PHP 等脚本语言，都属于主流语言。

这些语言在成长过程中都吸收了过去一些语言的优点，如 Java 的设计借鉴了 C++、Smalltalk 和 Lisp 的优点，而 Ruby 则是在吸收过去语言优点的基础上加以独自发展而形成的语言。

编程语言的进化方向

从过去编程语言的历史中，我们可以看出编程语言是在不断试错的过程中发展起来的。有很多编程语言已经消亡，仅仅在历史中留下了它们的名字，但其中所包含的思想，却被后来的语言以不同的形式吸取和借鉴。

例如，SNOBOL 的字符串处理功能，可以说是现代脚本语言基本功能的祖先。此外，20 世纪 70 年代由美国麻省理工学院（MIT）开发的一种名为 CLU[①]的语言中迭代器（Iterator）的概念，也被 Ruby 以代码块（Block）的形式继承了下来。

从编程语言的进化过程来看，一个显著的关键词就是“抽象化”。抽象化就是提供一个抽象的概念，使用者即便不具备关于其内部详细情况的知识，也能够对其进行运用。由于不必了解其内部的情况，因此也被称为“黑箱化”。

一些古老的编程语言，例如 BASIC 就没有实现充分的抽象化。虽然它提供了用于过程共享的子程序这个概念，但是子程序只能通过编号来调用，而且不能传递参数。由于“赋予名称”是抽象化的重要部分，所以说它的抽象化是不充分的。近代的编程语言中，都可以为一系列过程（程序）赋予相应的名称。

---

① CLU 是由 MIT 的芭芭拉 • 利斯科夫（Barbara Liskov，1939—　）所开发的语言，利斯科夫于 2008 年获得图灵奖。（原书注）

然而，仅仅将过程进行抽象化还远远不够。几乎所有的过程都需要进行一定的输入输出操作，而并不是与数据无关的。因此，在下一个阶段中，对数据进行黑箱化就显得非常重要。刚才我们提到的 CLU，就是数据抽象化出现早期的一种语言。

在数据抽象化的延长线上，就自然而然产生了面向对象编程的概念。所谓对象，就是抽象化的数据本身，因此面向对象和数据抽象化之间仅仅隔了薄薄的一张纸。在现在的 21 世纪编程语言中，面向对象已经是常识了，最近几乎所有的语言都或多或少地提供了面向对象的能力。当然，其中也有一些语言故意不提供对面向对象的支持①。

随着抽象化的不断深入，程序员即便不去关心内部的详细情况，也可以编写出程序。人类一次所能掌握的概念数量是有限的，有说法称，大部分人一次只能驾驭 7 ± 2 个左右的概念。这样一来，如果能够让问题的处理方式更加抽象，也就可以解决更复杂的问题。

受摩尔定律的影响，社会对于软件也提出了越来越高的要求。人类社会越来越依赖计算机，因此就需要开发出更多更可靠、更便宜的软件。

在讲述软件开发的一本名著《人月神话》中，作者弗雷德里克・布鲁克斯写道：②

> 无论使用什么编程语言，生产一条基本语句所需要的工数几乎是一定的。

也就是说，如果要描述同样的算法，A 语言需要 1000 行，B 语言只需要 10 行的话，只要采用 B 语言生产效率就可以提高 100 倍。

可能有人会觉得“这太扯了吧”。打个比方，用 Java 和 Ruby 描述同样的算法，语句行数相差 2 倍多也不稀奇，如果是汇编语言和 Ruby 相比的话，也许能产生 100 倍甚至 1000 倍的差距。

能产生这样的生产效率差异，正是抽象化的力量。抽象度高的编程语言不必描述详细过程，从而可以用简短的代码达到目的。和抽象化程度的差异相比，变量名称、有没有指定数据类型之类的都只能算是误差级别的差异而已。

---

① 例如，下一节中将要介绍的由保罗・格雷厄姆所设计的 Arc 语言，就（至少在其内置功能中）不支持面向对象，据保罗说，是有意这样设计的。（原书注）

②《人月神话——软件项目管理之道》(*The Mythical Man-Month: Essays on Software Engineering*)首次出版于 1975 年，并于 1995 年进行了扩充和再版。弗雷德里克・布鲁克斯(Frederick P. Brooks, Jr.，1931—)是美国的软件工程师，曾主持开发了 IBM 的 OS/360 操作系统，与 1999 年获得图灵奖。

未来的编程语言

从编程语言的进化这个视角来看，其实最近并没有什么大的动作。现在使用最广泛的编程语言几乎都是 10 多年前出现的，即便是比较新的 Java 和 Ruby 也是诞生于 20 世纪 90 年代后半，距离现在也已经是 15 年之前的事了。也许可以说，现在正是编程语言进化的好时机吧。

最近，受到 CPU 多核化等因素的影响，Erlang①这种并行处理语言受到了不少关注。不过 Erlang 早在 1987 年就诞生了，也并不是什么新东西，有点失望呀。

那么，未来的编程语言究竟会变成什么样呢?

美国风险投资家、Lisp 启蒙家、作家保罗・格雷厄姆在其《一百年后的编程语言》②一文中想象了 100 年后可能会出现的编程语言，并提议将他的观点应用到现在的编程语言中。

他主张，100 年后的编程语言进化的主线，应该以少量公理为基础的“拥有最小最简洁核心的语言”。在现有编程语言中，最具有这一特征的莫过于他最喜欢的 Lisp 了。所以说，他的主张实际上就是说，Lisp 才是 100 年后编程语言的进化方向。

唔，像我这样的小人物要跟他叫板好像也挺不自量力的，不过我还是认为，对于未来，应该基于从过去到现在的变化方向，并在其延长线上做出预测。当然，将来也许会发生一些无法预料的状况，从而大幅扭转之前的前进方向，不过这样的事情从定义来说本来就是无法预测的，你非要预测它，本质上也是毫无意义的。

作为一个编程语言御宅族，通过反观过去半个世纪以来编程语言的进化方向，我认为编程语言绝对不会按照保罗・格雷厄姆所说，向着“小而干净”的方向来进化。现在的编程语言，无论是功能上还是语法上都已经不是那样单纯了，虽然也曾经有人努力尝试将这些语言变得更小更简单，但包括保罗・格雷厄姆自己所设计的 Arc③在内，都决不能算是成功的尝试。

在我看来，编程语言的进化动机，不是工具和语言本身的简化，而是将通过这些工具和语言所得到的结果（解决方案）更简洁地表达出来。近半个世纪以来，编程语言不断提供愈发高

① Erlang 是由瑞典电信公司爱立信（Ericsson）旗下计算机科学研究室所开发的一种编程语言，发布于 1987 年，并于 1998 年实现开源。

② 保罗・格雷厄姆（Paul Graham，1964—　）是美国风险投资家、计算机科学作家。《一百年后的编程语言》（*The Hundred-Year Language*）一文收录于保罗・格雷厄姆的文集《黑客与画家》一书中，人民邮电出版社 2011 年 4 月出版，阮一峰译。

③ Arc 语言是 Lisp 的方言之一，由保罗・格雷厄姆与罗伯特・泰潘・莫里斯（Robert Tappan Morris，1965—　）共同设计，于 2008 年首次发布。

度的抽象化特性，也正是为了达到这个目的。因此我们可以很自然地认为，这种趋势在将来也应该会继续保持。

基于上述观点，如果要我来预测 100 年后编程语言的样子，我认为应该会是下面三种情况的其中之一：

(1) 变化不大。编程语言的写法从 20 世纪 80 年代开始就几乎没有什么进化，今后即便出现新的写法，也只是现有写法的变形而已。（从发展上来看，是比较悲观的未来）

(2) 使用编程语言来编程这个行为本身不存在了。人类可以通过和计算机对话（大概是用自然语言）来查询和处理信息。（类似《星际迷航》中的世界，对于编程语言家来说是比较失落的未来）

(3) 发明了采用更高抽象度写法的编程语言。这种语言在现在很难想象，不过应该是比现在更加强调 What，而对于如何解决问题的 How 部分的细节，则不再需要人类去过问。（难以预测的未来）

当然，上面的预测也只不过仅仅是预测而已，有可能与未来的实际情况大相径庭，或者说，与实际大相径庭的可能性比较大吧。不过话说回来，100 年后我也已经不在这个世上了，这不是白操心嘛。

20 年后的编程语言

通过对 100 年后的预测，我们明白了“预测 100 年后的事情是非常困难的”。想想看，100 年前连飞机还没有民用化呢，100 年后我已经可以坐在飞机上舒舒服服地写这本书的稿子了，这足以说明，要想象社会的变化是相当困难的。

那么，更近一点的未来又怎么样呢？比如说 20 年后。20 年前，日本刚刚改年号为平成[①]，现在和那个时候相比，印象中社会应该没有发生非常极端的变化。计算机的性能等方面确实有了长足的进步，不过发展趋势还是连续的，并非无法预测。对于 20 年后的未来，我想应该可以根据现在的发展趋势来做出判断。

个人认为，这么短的时间内，编程语言本身应该不会发生多大的变化。实际上，现在使用的很多语言，在 20 年前就已经存在的。因此我预计，20 年后的语言，应该是在分布处理（多

① 除了公元纪年外，日本人还普遍习惯使用年号纪年。和中国封建王朝时期一样，年号一般是随天皇的更换而进行更迭。1989 年，明仁天皇即位，改年号为平成，因此 2012 年是平成 24 年。在平成之前的上一个年号是昭和。

台计算机协作处理）和并行处理（多个 CPU 协作处理）功能上进行强化，使得开发者不需要特别花心思就能够使用这些功能。

之所以要关注分布处理和并行处理，是因为今后个人也可以通过云计算的形式使用到比现在更多的计算机，而随着每台计算机的 CPU 多核化，就相当于安装了更多的 CPU，这些情形都是很容易想象的。

不过，我认为现在的线程、RPC（Remote Procedure Call，远程过程调用）等显式地使用分布处理和并行处理的形式，早晚会遇到瓶颈。当核心数量超过数千个的时候，显式指定就变得毫无意义了，调试起来也会变得非常痛苦。我期待在 20 年后，能够出现突破这种局限的技术，即无需显式操作就可以实现分布处理和并行处理。

学生们的想象

几年前，我曾经在母校筑波大学开展过一次关于编程语言的集中讲座。在讲座中我给学生们出了“想象一下 20 年后的编程语言”这样一个题目，并在讲座最后一天提交报告。很有意思的是，大多数学生并没有做出我上面所说的关于分布处理和并行处理之类的技术性预测，而是提出了诸如“让编程变得更简单的语言”、“希望用自然语言来控制计算机”之类的想象。通过这些答案，似乎可以看出他们平常为了完成编程作业而被折磨得何等痛苦。

不过，这样的答案中，也许也蕴含着真理。近年来，编程语言似乎越来越难以脱离 IDE（Integrated Development Environment，集成开发环境）而单独拿出来说了。对于 Ruby 也总有人问“没有 IDE 吗？”之类的问题，当然，好消息是最近 Eclipse 和 NetBeans 已经支持 Ruby 了。

有点跑题了。总之，未来的编程语言可能不会像过去的编程语言那样，让语言本身单独存在，而是和编辑器、调试器、性能分析器等开发工具相互配合，以达到提高整体生产效率的目的。话说，那不就是 Smalltalk[①]吗？

唔，历史是否会重演呢？

① Smalltalk 正是一种考虑了“与编辑器、调试器、性能分析器等开发工具相互配合”而设计的语言。过去它并不能算是成功的，但随着技术的进步，其理念获得了越来越多的用武之地，或许真能卷土重来也未可知。（原书注）

2.2 DSL（特定领域语言）

下面，我们来介绍一些值得关注的编程语言功能。首先我们从 DSL（Domain Specific Language，特定领域语言）开始说起。

所谓 DSL，是指利用为特定领域（Domain）所专门设计的词汇和语法，简化程序设计过程，提高生产效率的技术，同时也让非编程领域专家的人直接描述逻辑成为可能。DSL 的优点是，可以直接使用其对象领域中的概念，集中描述“想要做到什么”（What）的部分，而不必对“如何做到”（How）进行描述。

例如，有一个名为 rake 的编译工具，它是用 Ruby 编写的。这个工具的功能跟 make 差不多，它会分析文件的依赖关系，并自动执行程序的编译、连接等操作。其中描述依赖关系的 Rakefile 就是使用了 Ruby 语法的一种 DSL。图 1 就是 Rakefile 的一个简单的例子。

```
task :default => [:test]
task :test do
  ruby "test/unittest.rb"
end
```

图 1　Rakefile 示例

这个例子表达了下面两个意思：

(1) 默认任务为 test
(2) test 的内容是执行 test/unittest.rb

启动 rake 命令的格式是：

```
rake 任务
```

如果这里的任务省略的话，则执行 default 任务。

在 Rakefile 对于依赖关系的描述中只是指定了 task，而对于内部数据结构是怎样的，以及维持依赖关系要如何实现等等具体问题都无需涉及，因为具体的实现方式，与描述依赖关系这个对象领域（Domain）是无关的。

DSL 这个对特定目的小规模语言的称呼，是最近才出现的比较新的叫法，但这种方法本身却并不是什么稀罕的东西，尤其是 UNIX 社区中就诞生了许多用来解释像这样的“专用语言”的工具。

其中，以行为单位进行文本处理的脚本语言 awk 算是比较有名的，除此之外，UNIX 中还开发了很多“迷你语言”，比如用来描述依赖关系的 Makefile 和用来读取它的 make 命令、以行为单位进行数据流编辑的 sed 命令、用来描述文档中嵌入图像的 pic 命令，用来生成表格的 tbl 命令等等。此外，为了对这些迷你语言的编写提供支持，UNIX 中还提供了语法分析器生成工具 yacc，以及词法分析器生成工具 lex，而 yacc 和 lex 本身也拥有自己的迷你语言。

这些迷你语言，几乎都是专用于特定用途的，大多数情况下无法完成复杂的工作，但它们都能够比简单的配置文件描述更多的内容，并为命令的处理带来了更大的灵活性，因此和 DSL 在本质上是相同的。

外部 DSL

像以这些迷你语言为代表的，由专用的语言引擎来实现的 DSL，称为外部 DSL。UNIX 的迷你语言文化是先于 DSL 出现的，但并非只有 UNIX 才提供外部 DSL。

在 Java 编写的应用程序中，大量使用了由可扩展标记语言（XML）编写的配置文件。这种 XML 配置文件也是外部 DSL 的一种形式。此外，数据库访问所使用的 SQL（Structured Query Language，结构化查询语言）也是一种典型的外部 DSL。

外部 DSL 的优点，在于它是独立于程序开发语言的。对于某个领域进行操作的程序，不一定是用同一种语言来编写的。例如，用来对数据库进行操作的程序，有时可以用 Ruby 来开发，有时也可以用 PHP 或者 Java 来开发。由于外部 DSL 是独立于程序开发语言的，因此可以实现跨语言共享。只要学会了 SQL，就可以在不同的语言中，用相同的 SQL 来进行数据库操作。

正则表达式的使用方法也差不多。正则表达式是用来描述字符串模板的一种外部 DSL，在 Ruby、Perl、PHP、Python 等很多语言中都可以使用，其语法在不同的语言中基本上都是通用的[①]。这样的好处是，在不同的语言中都可以使用字符串模板匹配这一通用功能。

此外，外部 DSL 实际上是全新设计的语言和语言引擎，因此可以根据它的目的进行自由的设计，不必被特定的执行模块和现有语言的语法所左右。由于自由度很高，在特定领域中能够大幅度提高生产效率。

也许大家认为设计和实现一种语言是非常辛苦的工作，但如果规模不是很大的话，实际上

① 实际上在不同引擎中，正则表达式的功能和语法也有细微差别。（原书注）

也没有那么难。以名著《UNIX 编程环境》[1]为首的许多书籍中，都以迷你语言的制作作为例题，其核心部分只要几页的代码就可以完成。

然而，高度自由的设计也是一把双刃剑，这意味着程序员为了使用 DSL 必须学习一门全新的语言。像 SQL 这样作为数据库访问的通用方式已经实现普及和标准化的语言还好，如果为了每一种目的都要从零开始学习一门完全不同的语言，这样的辛苦可不是闹着玩的。

内部 DSL

和外部 DSL 相对的自然就是内部 DSL 了。外部 DSL 是从 UNIX 中脱胎发展而来的，而内部 DSL 则是发源于 Lisp 和 Smalltalk 的文化之中。

内部 DSL 并不是创造一种新的语言，而是在现有语言中实现 DSL，而作为 DSL 基础的这种现有语言，称为宿主语言。从原理上说，宿主语言可以是任何编程语言，不过还是有适合不适合之分。Lisp、Smalltalk、Ruby 这些语言适合作为 DSL 的宿主语言，因此在这些语言的社区中经常使用内部 DSL。

内部 DSL 的优点和缺点和外部 DSL 正好相反。也就是说，内部 DSL 是“借宿”在宿主语言中的，它借用了宿主语言的语法，因此程序员无需学习一种新的语言。在理解内部 DSL 含义时，宿主语言的常识依然有效，而学习新语言时，宿主语言的知识也会成为不错的航标。

之前我们举过 rake 命令的 Rakefile 这个例子，这就是一种内部 DSL。图 1 中显示的 Rakefile 代码，是用来为 rake 命令描述编译规则的 DSL 代码，但同时它也是一段 Ruby 程序。

内部 DSL 还有其他一些优点。当用 DSL 编写复杂的逻辑时，可以使用过程定义、条件分支、循环等作为通用语言的宿主语言所具备的全部功能。从某种意义上说，它就是万能的。

此外，“寄生”在宿主语言上面，就意味着 DSL 的实现会相对容易。一种迷你语言的实现再怎么简单，和在宿主语言基础上增加一些功能就能够实现的内部 DSL 相比，都只能甘拜下风了。

说到内部 DSL 的缺点，由于 DSL 的语法被限定在宿主语言能够描述的范围内，因此自由度比较低。不过，自由度低换来了较好的易读性，何况像 Lisp 这样具备高性能宏功能的语言中，即便是内部 DSL（以一定的易读性为代价）也可以实现相当高的自由度。

我的个人观点是，从易读性和实现的容易性来看，内部 DSL 具备更多的优势。

① 《UNIX 编程环境》（*The Unix Programming Environment*），Brain W. Kernighan 和 Rob Pike 著，1984 年出版，两位作者来自贝尔实验室，这本书是关于 Unix 操作系统的早期著名著作。中译本由机械工业出版社于 1999 年发行，陈向群等译。

DSL 的优势

那么 DSL 有哪些优势呢？为什么 DSL 近年来如此备受关注呢？

这是因为 DSL 在几个方面上可以说掌握了提高生产效率的关键。DSL 拥有为特定领域所设计的词汇，可以在高级层面上编写程序。由于不需要编写多余的部分，因此就节约了程序开发的时间。

此外，使用 DSL 可以让程序在整体上以更简洁的形式进行表达，这意味着无论是写程序还是读程序的成本都降低了，同时也意味着对于非编程专家的一般人来说，编程的大门正向他们敞开。

很多人觉得编程很难，但如果自己的专业领域中有适用的 DSL 的话，情况就不同了。如果可以将想让计算机完成的任务直接描述出来，也许就可以减少与程序员交流的成本，从而实现生产效率的提高。这才是 DSL 备受关注的一个最大的理由。

仔细想想就会发现，不涉及对象领域的内部细节，而是在高级层面上进行描述，这就是近半个世纪以来编程语言进化的方向——抽象化。也就是说，DSL，尤其是内部 DSL，也许就是由抽象化的不断推进所引领的编程语言未来发展的方向之一吧。

DSL 的定义

谈到 DSL，大家总是热衷于讨论到底怎样算是 DSL。关于什么是 DSL，什么不是 DSL，并没有明确的定义和标准。

有一种观点认为，像“是否具备仅用于特定用途的功能”、“（设计者）自己是否将其命名为一种 DSL”等可以作为判断的标准，但实际上这些标准也是非常模棱两可的。

尽管如此，考虑到 DSL 实际上是编程语言抽象化的延伸，那么问题就不应该是什么是 DSL、什么不是 DSL，DSL 应该是将面向特定领域的 API 设计成优秀的 DSL 这样一个设计的过程。

据说，在诞生了 UNIX 的 AT&T 贝尔实验室[①]中有一句名言：库设计就是语言设计（Library design is language design）。我们在思考编程语言的时候，大多仅强调语法，但如果脱离了相当于词汇的库、类和方法，语言也就无从思考。

① 贝尔实验室（Bell Laboratories）是一个以电信相关研究起家的研究开发机构，由 AT&T 和 Western Electric 共同出资创立于 1925 年，于 1996 年脱离 AT&T，目前属于阿尔卡特朗讯（Alcatel-Lucent）公司旗下。贝尔实验室在计算机领域的重大发明包括 UNIX 操作系统和 C 语言等，其他领域的重大发明包括晶体管、激光、CCD 等，许多成果都获得过诺贝尔奖。

也就是说，API（Application Programming Interface，应用程序编程接口）也是构成编程语言的一个重要要素。向一种语言添加库，也就相当于在设计一种“新增了一些词汇的规模更大一点儿的语言”。我们可以通过“编程达人”大卫·托马斯[①]的这句话理解这一过程：

Programming is a process of designing DSL for your own application.

（编程就是为自己的应用程序设计 DSL 的过程）

应用程序开发，可以理解为是将库等组件设计成针对该应用程序对象领域的 DSL，最后再进行整合的过程。这样编写出来的应用程序，其代码的抽象度高，应对未来修改的能力强，一定是一个不错的应用程序。

因此，DSL 并不仅仅是一种技术，而是应用程序开发的重要设计原理和原则之一，可以说适用于任何软件的开发。在设计 API 时，如果能像“设计一种新的 DSL”一样进行设计的话，感触应该会变得不同吧。

适合内部 DSL 的语言

正如刚才所说的，在 UNIX 文化中，由若干单一目的的小工具所组成的“工具箱系统”是主流，Linux 也继承了这一点。UNIX 中的各种迷你语言，作为组成工具箱的零部件，同时也作为为特定目的的专用的外部 DSL，不断发展壮大起来。

不过，在现代 UNIX 文化也受到了很多来自外部的影响。例如，现在典型的 UNIX 文本编辑器 Emacs，其起源与其说是来自 UNIX，不如说是来自美国麻省理工学院（MIT）的 Lisp 文化。它的开发者理查德·斯托曼[②]本来就是一位 MIT 出身的 Lisp 黑客，因此这也是理所当然的吧。此外，从 Perl 到 Ruby 的这些脚本语言中，并不是采用小工具组合起来的方式，而是提供多功能可编程的工具，从这一点上看，也是受到了 Lisp 文化的影响。近年来内部 DSL 的备受关注，和这个倾向也不能说没有关联。

那么，什么样的语言适合用作内部 DSL 的宿主语言呢？虽然任何语言都可以成为宿主语言，但像 Lisp、Smalltalk、Ruby 这样被认为适合 DSL 的语言，都拥有一些共同的特征。

首先是简洁。由于 DSL 本来就是为了将针对特定目的的处理用高级的、简洁的方式进行描述，

① 大卫·托马斯（Dave Thomas）是一位程序员、作家、出版家，著有 *Programming Ruby* 等书。
② 理查德·斯托曼（Richard Stallman，1953— ）是一位著名的黑客，自由软件运动领导者，GNU 计划和自由软件基金会创始人，他所编写的 GNU 通用公共许可证（GNU GPL）是世界上最广为采用的自由软件许可证。

因此简洁的描述方式才是最本质的。从这个意义上来讲，语言简洁是作为 DSL 宿主语言不可或缺的要素。Lisp 和 Ruby 等语言中，无需在程序中声明数据类型，编译器的“规矩”也比较少，因此能够让程序变得简洁。

作为宿主语言的另一个重要特征就是灵活性。在 DSL 中，开发者会通过高度抽象化的代码来集中描述 What，而不是 How，因此作为对抽象化的支持，元编程等功能也最好比较充实。

此外，Lisp 具备宏（Macro）功能，只要遵循“用括号进行表达”的 S 表达式[①]语法，就可以实现相当自由的表达。因此可以说 Lisp 语言本身就是一种可被编程的语言。

另一方面，Ruby 中的代码块（Block）功能可以实现控制结构。代码块虽然不像 Lisp 的宏那样万能，但用来实现内部 DSL 还是足够了。

让我们再回头看看图 1 中 Rakefile 的示例。在 Rakefile 中定义了一个名为 task 的方法，任务的名称则作为参数通过符号（Symbol）来指定。当这里定义的 test 任务被执行时，代码块就会被求值。像这样，在 Ruby 中通过使用代码块，就可以表达一种控制结构。

Rakefile 的代码之所以看上去很简洁，是因为 Ruby 的语法就是以这样的宗旨进行设计的。相比语法的单纯性来说，Ruby 更加重视程序的可读性，其语法也是以此为先决条件而确定的。

例如，调用方法时参数周围的括号是可以省略的，还可以通过代码块将整个一块代码作为参数传递给一个方法。如果说 Ruby 的语法追求的是没有任何冗余性的“简单”的话，那么图 1 的 Rakefile 代码就会变成图 2 这个样子。这样的语法虽说非常简单，但绝对算不上是易读和易写的。

```
task(:default => [:test])
task(:test, &lambda(){
  ruby("test/unittest.rb")
})
```

图 2　Rakefile 示例

那么，如果是一种不具备 Lisp 和 Ruby 这样简洁性和灵活性的语言，例如 Java，是不是就不可能用作内部 DSL 呢？的确，像 Java 这样的语言，由于必须要指定数据类型，代码容易变得非常繁杂，而且语法的自由度也不高，要实现像 Lisp 和 Ruby 一样的内部 DSL 是非常困难的。然而，以代码重构而闻名的马丁 • 福勒[②]则提出，通过“流畅接口”（Fluent interface）的方式，像 Java 这样的语言是不是也能够实现内部 DSL 一样的功能呢？图 3 就是他所展示的流畅接口的示例。

① S 表达式（S-expression）是符号化表达式（Symbolic Expression）的简称，是 Lisp 中用于表达嵌套（层次状）数据的方式。

② 马丁 • 福勒（Martin Fowler，1963—　）是一位软件开发方面的著名作家，这里所说的“代码重构”是指他在 1999 年与肯特 • 贝克等人合作编写的《重构：改善既有代码的设计》（*Refactoring: Improving the Design of Existing Code*）一书。

图 3 中的代码定义了一张顾客（Customer）的订单。在订单中，包含了商品及其数量的订购明细。某些情况下，为了避免整张订单延迟发货，可以从订单中去掉某些货品，先将剩下有货的货品发出来，因此在这里的明细中，有些货品被定义为“赶不上货期的话可以跳过”。整张订单的状态被设定为“加急”。图 3 整个代码的含义是定义了如下这样一张订单：

```
private void makeOrder(Customer
customer) {
    customer.newOrder()
            .with(6, "TAL")
            .with(5, "HPK").skippable()
            .with(3, "LGV")
            .priorityRush();
}
```

图 3　用 Java 编写的流畅接口（fluent interface）

- ❑ TAL，6 个
- ❑ HPK，5 个（可跳过）
- ❑ LGV，3 个
- ❑ 加 急

流畅接口中运用了方法链（Method chain），作为 Java 来说这种表达方式是前所未有的简洁。如果用以前“更像 Java 的风格”来表达的话，就会变成图 4 这样。

简洁性和易读性的区别一目了然。虽然流畅接口在 Java 社区还没有普及，不过这种设计思路在今后是非常值得期待的。

```
private void makeOrder(Customer customer) {
    Order o1 = new Order();
    customer.addOrder(o1);
    OrderLine line1 = new OrderLine(6, Product.
find("TAL"));
    o1.addLine(line1);
    OrderLine line2 = new OrderLine(5, Product.
find("HPK"));
    o1.addLine(line2);
    OrderLine line3 = new OrderLine(3, Product.
find("LGV"));
    o1.addLine(line3);
    line2.setSkippable(true);
    o1.setRush(true);
}
```

图 4　Java 标准风格的接口

外部 DSL 实例

内部 DSL 代表了编程语言进化的一种形态，作为编程爱好者，我自然对其兴趣颇深，但在这里，我还想再谈谈外部 DSL。

刚才已经讲过，外部 DSL 就是拥有专用引擎的一种独立的特定领域语言，不过外部 DSL 也有各自不同的实现水平。

最简单的一种莫过于配置文件和数据文件了吧。例如 YAML、JSON 等语言，就是为了“将对象（用对人类易读的形式）描述出来”这一特定目的而设计的外部 DSL（图 5）。

```
[
  {
    "name": "松本行弘",
    "company": "NaCl",
    "zipcode": "690-0826",
    "address": "松江市学园南2-12-5",
    "tel": "0852-28-XXXX"
  }
]
---
```

图 5　外部 DSL JSON 中对数据的表达方式

而更复杂的一些 DSL，虽然也是为特定目的而设计，但却可以编写出描述任意算法的程序。例如用于文本处理的 awk 等，就属于这种水平。awk 程序的基本结构是从文件中以行为单位读取字符串，同时它具备每读取一行之后将字符串分割成字段等等这样的文本处理专用功能。这些功能明显是属于 DSL，但另一方面，它也并不是不能编写出文本处理范围以外的程序。再举一个例子，用于向打印机描述页面的 DSL PostScript，也是一种基于逆波兰记法[①]的图灵完全（拥有完整计算能力的）语言。

另外还有一种比较特殊的，虽然它本身是一种通用语言，叫做 DSL 并不十分合适，但这种语言却与特定计算模型之间拥有很强的关联性，也就具备了很强的类似 DSL 的性质。例如 Prolog，它是一种以一阶谓词逻辑为基础的语言。Prolog 可以被认为是面向谓词逻辑这一特定领域的 DSL，但将谓词逻辑这个适用范围称作“特定领域”似乎未免太宽泛了，因此一般情况下人们也不会将 Prolog 称作 DSL。同样地，与用于并行计算的 Actor 模型[②]密切相关的 Erlang 等语言，虽然是一种通用语言，但它同时也具备“面向并发编程领域的 DSL”这一性质。

让我觉得比较有意思的是 Java 中所使用的 XML。由于 Java 中默认内置了用于解析 XML 的库，因此如果用 XML 来编写 DSL 的话，就可以很容易地被程序读取。这样一来，基本上就可以省却为 DSL 开发语言引擎的步骤了。

通过这样的方式，我们可以用 XML 这一通用的、不被特定目的限制的语法，很容易地创造出新的外部 DSL，我认为这是一种非常高效的方式。只不过，XML 文件的内容是通过标签来描述的，看起来十分冗长，无论是阅读还是编写，对用户来说都不是很友好，这一点算是一个遗憾吧。

DSL 设计的构成要素

曾经在诸多 Ruby 相关活动中发表过演讲的著名 Rubyist——Glenn Vanderburg 认为，构成一种优秀的（内部）DSL 的要素包括下列 5 种：

① 逆波兰记法（Reverse Polish notation）是一个将操作符置于操作数后面的“后缀式记法”，如“3 + 4”用这种记法要写成“3 4 +”。这种记法的好处是不必使用括号来表示运算顺序，并且对于计算机来说更容易实现。

② Actor 模型（Actor model）是一种用于并行计算的模型，其中每个 Actor 是并行计算的基本单元，它可以和其他 Actor 进行消息通信，也可以创建更多的 Actor。

- ❑ 上下文（Context）
- ❑ 语句（Sentence）
- ❑ 单位（Unit）
- ❑ 词汇（Vocabulary）
- ❑ 层次结构（Hierarchy）

其中的上下文，用来针对 DSL 中每个单独的语句，规定其所拥有的含义。也许有人认为：“用参数的方式进行显式指定不就好了吗？”不过大家别忘了，DSL 的宗旨是进行简洁的描述，如果每次都通过参数来反复指定上下文的话，程序必定会变得冗长。

请看图 6 中的程序。这是用描述测试用的库 shoulda 来编写的测试程序，它也是一种以 Ruby 为基础的内部 DSL。

```
class UserTest < Test::Unit::TestCase
  context "a User instance" do
    setup do
      @user = User.find(:first)
    end

    should "return its full name" do
      assert_equal 'John Doe', @user.full_
name
    end
  end
end
```

图 6　shoulda 编写的测试程序

在图 6 的例子中，context 方法和 should 方法就定义了上下文。顾名思义，context 方法的作用就是定义上下文，它表示这个测试项目的一个大的框架，而上下文的范围是通过指派给 context 方法的代码块来定义的。因此：

```
context "a User instance" do
```

就表示“对 a User instance 这个上下文的测试进行定义”的意思。

should 方法则定义了“需要满足某个条件”这样一个测试。当指派给 should 方法的代码块所描述的测试成功时，则视为满足该测试的条件。should 方法所定义的测试，和外部的上下文结合起来，就定义了“a User instance should return its full name”（用户实例应当返回其全名）这一软件的行为。

这样的方式与其说是测试，不如说是定义了软件所应该具有的行为（Behavior），因此更多的情况下人们不会将其称为测试，而是称为规格（Spec）。此外，在软件开发之前就设计好规格的开发手法，通常不是叫做测试驱动开发，而是叫做行为驱动开发（Behavior Driven Development，BDD）。

说完了上下文，下一个 DSL 的构成要素是“语句”。语句也就是上下文中每条独立的代码，在内部 DSL 中实现函数和方法的调用。

貌似英语圈的人总是把让语句尽可能看起来接近英语这一点看得很重要。DSL 的优点之一，

就是让并非专家的普通人也能够使用，因此，为不太懂编程的人降低点门槛，这样所带来的好处也是可以理解的。不过，作为我来说（也是因为我英语不是很好的缘故吧），我觉得看上去像英语并不是 DSL 的本质，不过还是有很多人执着于这一点。

在几年前的一次 Ruby 大会中有一个以 DSL 为主题的段落，讲的内容基本上都是围绕着“在 Ruby 语法的范围内到底能设计出多接近英语的 DSL”这个话题，从各种角度来说都让我觉得很有意思。例如，如果你问我，一个用来描述面包做法的 DSL 写成像图 7 这样好不好，说实话我还真没有什么自信。

```
recipe "Spicy Bread" do
  add 200.grams.of Flour  # 加入200克小麦粉
  add 1.lsp.of Nutmeg     # 加入一大勺肉豆蔻
  #继续……
end
```

图 7　描述面包做法的 DSL

话说，如果忽略符号的话，作为英语来理解也确实没有太大问题，但越是接近自然语言，作为外行人就越容易纠结在一些微妙的差异上，例如到底要在哪里加上符号，顺序能不能调换等等。我觉得这正是 COBOL 曾经经历过的坎坷,但英语圈的人们却似乎有着不同的感触和理解。

接下来的一个要素是单位，也就是在图 7 的例子中出现的克（grams）、大勺（lsp）等等。

由于在 Ruby 中可以在已有的类中自由添加新的方法，利用这一点，在上面的例子中实际上是在整数类中定义了 grams 方法和 lsp 方法。

我第一次见到这样的扩展功能，是在 Ruby on Rails[①]所包含的 ActiveSupport 库中。当时我看到“现在时间的 20 小时之前”居然能够写成“20.hours.ago”的时候感到非常震惊。实际上，是整数类中的 hours 方法对 20.hours 这一调用返回了 72000（3600 秒 ×20）这个值，而 ago 方法又返回了表示该秒数之前这一时刻的一个 Time 对象而已。

这样看来，Rails 不仅是一种 Web 应用程序框架，同时也可以说是以 Web 应用程序开发为对象领域的，以 Ruby 为基础的内部 DSL。

Glenn Vanderburg 所说的另外两个要素就是词汇和层次结构。前者的意思是，为目的领域定义了多少适用的方法，对必要方法的自动生成功能也包含在内。

例如，在 Rails 中，如果数据库的 users 表中包含 name 这一属性的话，那么就可以进行这样的调用：

① Ruby on Rails 简称 Rails，是用 Ruby 语言编写的一个非常有名的开源 Web 开发框架，它遵循 MVC 模式，提出了“不做重复的事”、“惯例优于配置”等理念，这些理念也被后来其他语言编写的一些 Web 开发框架（如 CakePHP、Zend 等）借鉴。

```
User.find_by_name("松本")
```

其中 find_by_name 方法就是自动生成的。

层次结构可以理解为嵌套的上下文。

Vanderburg 举了图 8 这样的一个例子。这是使用 XmlMarkup 库的一个例子，将 Ruby 作为一种 DSL 来生成 XML，看起来可能比 XML 的代码要易读易写得多。由图 8 的代码，用 XmlMarkup 所生成的 XML 文件的内容如图 9 所示。

```
xml = Builder::XMLMarkup.new
xml.html {
  xml.head{
    xml.title("History")
  }
  xml.body {
    xml.h1("Header")
    xml.p("paragraph")
  }
}
```

图 8　用 XmlMarkup 这一 DSL 来生成 XML

```
<html>
  <head>
    <title>History</title>
  </head>
  <body>
    <h1>Header</h1>
    <p>paragraph</p>
  </body>
</html>
```

图 9　XmlMarkup 所生成的 XML

Sinatra

我觉得 Rails 具备 DSL 的性质并不是有意为之，而是“为提高生产效率而追求抽象化而水到渠成的结果”，但比 Rails 更新的 Web 应用程序框架中，出现了一些明显对 DSL 有所意识的项目，其中的代表就是 Sinatra。

Sinatra 是定位于较小规模 Web 应用程序的框架，用 Sinatra 编写的用来显示“Hello, world!”字符串的应用程序如图 10 所示。对于 Web 浏览器传送过来的“get /”请求应当如何响应，正是用 DSL 的形式来表达的。这种十分简洁的表达方式，以及应用了上下文和语句的代码风格，和 Rakefile、shoulda 这些以 Ruby 为基础的内部 DSL 可以说是有异曲同工之妙。

```
# hello_world.rb
require 'sinatra'

get '/' do
  "Hello world, it's #{Time.now} at the server!"
end
```

图 10　用 Sinatra 编写的 Web 版 Hello World

小结

DSL 是在 Ruby 界备受关注的一种方式，或者可能有点红过了头。DSL 这个称呼虽然只是最近才出现的，但实际上它已经是从数十年前开始就已经在使用的技术了。

不过，正如通过给一种设计模式起个名字能够提高其认知度，从而让更多的人来使用它一样，DSL 也可能因为被赋予了这个名字而获得了广泛的认知，从而对编程生产效率的提高做出贡献。今后，DSL 也可能作为判断设计优秀与否的重要指标，对软件开发产生巨大的影响。

2.3 元编程

在 Ruby 界中，元编程再度成为一个热点话题。Ruby on Rails 等框架中，其生产性就是通过元编程来实现的，这一点其实已经是老生常谈了。不过，《Ruby 元编程》[①]一书于 2010 年由 ASCII Media Works 出版了日文版，因此貌似有很多人是通过这本书才重新认识元编程的。

在 2010 年的 RubyKaigi[②]上，有幸请到了《Ruby 元编程》一书的作者 Paolo Perrotta 来日本演讲，我跟他也简单聊了聊，他好像并没有 Lisp 的经验，这令我感到非常意外。那么我们就一边参考作为原点的 Lisp，一边来重新审视一下元编程吧。

Meta, Reflection

"元"这个词，是来自希腊语中表示"在……之间、在……之后、超过……"的前缀词 meta，具有超越、高阶等意思。从这个意思引申出来，在单词前面加上 meta，表示对自身的描述。例如，描述数据所具有的结构的数据，也就是关于数据本身的数据，被称为元数据（Metadata）。再举个比较特别的例子，小说中的角色如果知道自己所身处的故事是虚构的，这样的小说就被称为元小说（Metafiction）[③]。

综上所述，我们可以推论，所谓元编程，就是"用程序来编写程序"的意思。那么，用程序来编写程序这件事有什么意义吗？

像 C 这样的编程语言中，语言本身所提供的数据，基本上都是通过指针（地址）和数值来表现的。在语言层面上虽然有数组和结构体的概念，但经过编译之后，这些信息就丢失了。

不过，"现代派"的语言在运行的时候，还会保留这样一些信息。例如在 C++ 中，一个对象是知道自己的数据类型的，通过这个信息，可以在调用虚拟成员函数时，选择与自己的类型（类）相匹配的函数。在 Java 中也是一样。

---

① 《Ruby 元编程》（*Metaprogramming Ruby*），Paolo Perrotta 著，原书出版于 2010 年 2 月，中文版于 2012 年 1 月由华中科技大学出版社出版，廖志刚、陈睿杰译。

② RubyKaigi，原名"日本 Ruby 会议"，是一项关于 Ruby 编程语言的集会活动，从 2006 年开始每年在日本举办。

③ 更常用的中文译法是"后设小说"，也叫"超小说"、"自反小说"。

像这样获取和变更程序本身信息的功能，被称为反射（Reflection）。将程序获取自身信息的行为，用“看着（镜子中）反射出的身影来反省自己”这样的语境来表达，听起来还挺文艺的呢。

和 Java、C++ 等语言相比，在 Ruby 中大部分信息都可以很容易地进行访问和操作。作为例子，我们从定义一个类并创建相应的对象开始看。

```
class Duck
  def quack
    "quack"
  end
end
duck = Duck.new
```

我们用 class 语句定义了一个名为 Duck 的类。在 Ruby 中，类也是一个普通的对象，也就是说：

```
duck = Duck.new
```

表示调用 Duck 类这个对象中的 new 方法，而 new 方法被调用的结果，就是生成并返回了一个新的 Duch 类的实例。

在生成出来的实例中，包含有作为其“母版”的类的信息，这些信息可以通过调用 class 方法来查询。

```
duck.class   # => Duck
```

当调用实例（即对象）的方法时，实例会去寻找自己的类。

```
duck.quack  # => "quack"
```

换句话说，当调用 duck 的 quack 方法时，首先要找到 duck 的类（Duck），然后再找到这个类中所定义的 quack 方法。于是，由于 Duck 类中定义了 quack 方法，因此我们便能够调用它了。

Duck 类中定义的方法只有 quack 一个，我们来确认一下。调用类的 instance_methods 方法可以得到该类中定义的方法一览。

```
Duck.instance_methods(false)
# => [:quack]
```

参数 false 表示仅显示该类中定义的方法。如果不指定这个参数，Duck 类会将从它的父类(超类）中继承过来的方法也一起显示出来。

在调用方法时，如果这个方法没有在类中定义，则会到超类中去寻找相应的方法。举个例子，请看下面的代码。

```
duck.to_s # => "#<Duck:0xb756ed2c>"
```

由于 Duck 类中并没有定义 to_s 方法，因此要到 Duck 类的超类中去寻找。可是，Duck 类的超类又是什么呢？在定义 Duck 类的时候我们并没有指定超类。

这样的信息，问问 Ruby 就能获得最准确的答案。用 ancestors 方法就可以得到相当于该类“祖先”的超类一览。

```
Duck.ancestors
# => [Duck,Object,Kernel,BasicObject]
```

从这里我们可以看出，Duck 类的超类是 Object 类。在 class 语句中如果不指定超类的话，则表示将 Object 类作为超类。

那么，to_s 是否在 Object 类中进行了定义呢？

```
Object.instance_methods(false)
# => []
```

咦，Object 类中一个方法都没有定义。其实，像 to_s 这样所有对象都共享的方法，是在其上层的 Kernel 模块中进行定义的。

到此为止，对象的结构如图 1 所示。因此，在 Ruby 中，即便是像类这样的元对象，也可以像一般的对象一样进行操作。不对，“像……一样”这个说法还不准确，在 Ruby 中，类和其他的对象是完全相同的，没有任何区别。

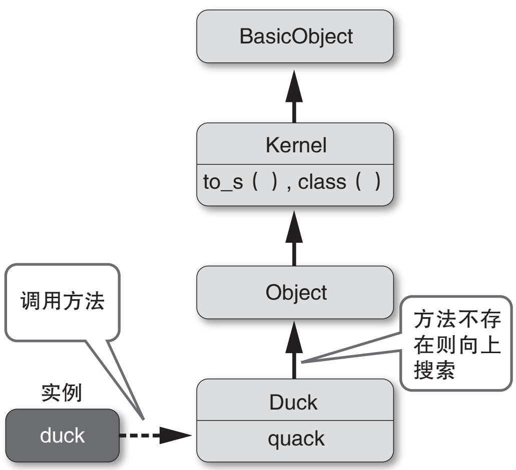


图 1　对象结构图（Ver.1）

类对象

可是，如果说类和其他的对象完全没有任何区别的话，那么类的类又是什么呢？我们还是来问问 Ruby 吧。

```
Duck.class # => Class
```

也就是说，在 Ruby 中有一个名叫 Class 的类，所有的类都可以看做是这个 Class 类的实例。有点像绕口令呢。

再刨根问底一下，如果所有的类都是 Class 类的实例，那么 Class 类的类又是什么？

```
Class.class # => Class
```

Class 类的类居然是 Class，也就是它自己本身，这真是出乎意料呢。我们再来看看 Class 的超类又是什么吧。要查看类的直接上一级父类，可以使用 superclass 方法。

```
Class.superclass # => Module
```

原来 Class 的超类是 Module 呢，而 Module 的超类则是 Object。

```
Module.superclass # => Object
```

将上面所有的信息综合起来，更新之后的对象结构图如图 2 所示。像这样，就可以直观地看到 Ruby 中对象和类所具有的层次结构了。

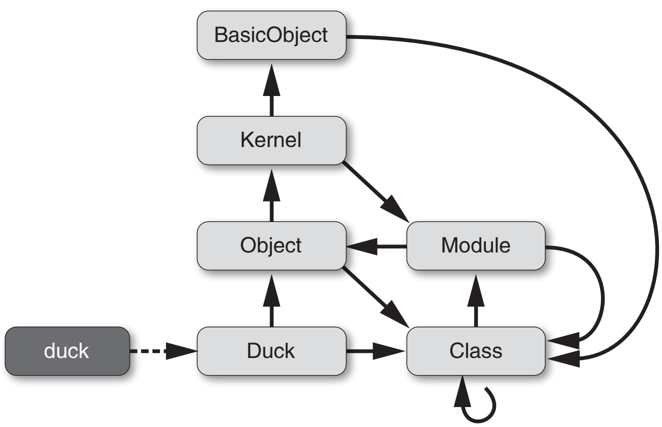


图 2　对象结构图（Ver.2）

类的操作

不过，即便说类也是一种对象，但类的定义和方法的定义都有专用的“语法”，很难消除这种 Ruby 独有的特殊印象。我们来看看上面讲过的 Duck 类的定义。

```
class Duck
  def quack
    "quack"
  end
end
```

其实，和 Ruby 的其他部分一样，通过方法的调用也可以实现同样的操作。

```
Duck = Class.new do
  define_method :quack do
    "quack"
  end
end
```

怎么样？“先创建一个 Class 类的新实例，然后赋值给 Duck 常量”，“对 quack 方法进行定义”，通过这样的步骤，是不是有更直接的感触呢？不过，这种写法可实在算不上是易读，我也不会推荐大家这样写，这种写法只是能够帮助你更直观地理解“Ruby 的 Class 也完全是一个普通的 Ruby 对象”这一概念而已。好，我们来仔细看看 define_method 的部分。上面代码中的第 2 行到第 4 行内容，和下面的 def 语句是等同的。

```
def quack
  "quack"
end
```

不愧是有专用语法的 def，代码就是简洁。这里的 define_method 方法所执行的具体操作如下。

- ❑ 以参数的方式接收表示方法名的符号（:quack）
- ❑ 以代码块的形式接收方法的定义部分

运行这个方法的实体又是什么呢？先透露一下答案吧，运行这个方法的主体就是 Duck 类。在方法定义的部分（class 的代码中、Class.new 的代码块中）中，self 所表示的是现在正在被定义的类。在这样的定义中，类就可以通过调用 define_method 方法来向自身添加新的方法。

不过，肯定有人会说，用这种比 def 更加冗长的方法到底有什么意义呢？因为通过调用方法来定义方法，可以为我们打开新的可能性。

例如，假设在某种情况下需要定义 foo0 ～ foo99 这样 100 个方法，在程序中写 100 个 def 语句实在是太辛苦了。如果是 Java 的话，通过代码生成器，也许可以用不着真的看到那 100 个

方法的定义。而在拥有 define_method 的 Ruby 中，我们用下面这样简单的程序就可以定义 100 个方法。

```
100.times do|i|
  define_method("foo#{i}") do
    ...
  end
end
```

用一个循环来代替 100 个定义，这才是 define_method 真正的用武之地。

这个方法中 self 表示现在正在定义的类，利用这一性质，在 Ruby 中可以实现各种各样的操作。例如：

- ❑ 指定要定义的方法在怎样的范围内可见（public, protected, private）
- ❑ 定义对实例变量的访问器（attr_accessor, attr_reader, attr_writer）

像这样，本来属于“声明”的内容，通过调用方法就可以实现，这一点是 Ruby 的长处。

Lisp

拥有这方面长处的语言并不只有 Ruby，Lisp 可以说是这种语言的老祖宗。Lisp 的历史相当悠久，其诞生可以追溯到 1958 年。说起 1958 年，在那个时候其他的编程语言几乎都还没有出现呢。在那个时代已经存在，并且现在还依然健在的编程语言，也就只有 FORTRAN（1954 年）和 COBOL（1959 年）而已了吧。Lisp 作为编程语言的特殊之处，在于它原本并不是作为一种编程语言，而是作为一种数学计算模型设计出来的。Lisp 的设计者约翰・麦卡锡，当时并没有设想过要将其用作一种计算机语言。麦卡锡实验室的一名研究生——史蒂芬・罗素，用 IBM 704 的机器语言实现了原本只是作为计算模型而编写的万能函数 eval，到这里，Lisp 才真正成为了一种编程语言。

Lisp 在编程语言中可以说是类似 OOPArts[①]一样的东西。编程语言的历史是由机器语言、汇编语言开始，逐步发展到 FORTRAN、COBOL 这样的“高级语言”的。而在这样的历史中，作为最古老语言之一的 Lisp，居然一下子具备了超越当时时代的很多功能。

1995 年 Java 诞生的时候，虚拟机、异常处理、垃圾回收这些概念让很多人感到耳目一新。从将这些技术普及到“一般人”这个角度来说，Java 的功绩是相当伟大的。但实际上，所有这

① Out Of Place Artifacts 的缩写，意思是“与时代不符的（使用了先进技术的）遗物”。（原书注）

些技术，早在 Java 诞生的几十年前（真的是几十年前），就是已经在 Lisp 中得到了实现。很多人是通过 Java 才知道垃圾回收的，而 Lisp 早期的解释器中就已经具备了垃圾回收机制。由于在 Lisp 中数据是作为对象来处理的，内存分配也不是显式指定的，因此垃圾回收机制是不可或缺的。于是这又是一项 40 多年前的技术呢。

像虚拟机（Virtual machine）、字节码解释器（Bytecode interpreter）这些词汇，也是通过 Java 才普及开来的，但它们其实是 Smalltalk 所使用的技术。Smalltalk 的实现可以追溯到 20 世纪 70 年代末到 80 年代初，因此这一技术也受到了 Lisp 的影响，只要看看就会发现，Smalltalk 的解释器和 Lisp 的解释器简直是一个模子刻出来的。

数据和程序

凡是看过 Lisp 程序的人，恐怕都会感慨“这个语言里面怎么这么多括号啊”。图 3 显示的就是一个用于阶乘计算的 Lisp 程序，图 4 则是用 Ruby 写的功能相同的程序，大家可以比较一下，括号的确很多呢，尤其是表达式结束的部分那一大串括号，相当醒目。这种 Lisp 的表达式写法，被称为 S 表达式。不过，除此之外的部分基本上是可以一一对应的。值得注意的有下面几点：

- Lisp 是通过括号来体现语句和表达式的
- Lisp 中没有通常的运算符，而是全部采用由括号括起来的函数调用形式
- “1-”是用来将参数减 1 的函数

```
;;; 通过归纳法定义的阶乘计算
(defun fact(n)
  (if (= 1 n)
     1
    (* n (fact (1- n)))))

(fact 6)  ;; => 结果为720
```

图 3　Lisp 编写的阶乘程序

```
# 这里体现了Lisp和Ruby的相似性
def fact(n)
  if n == 1
    1
  else
    n * fact(n - 1)
  end
end

fact(6)  # => 结果为720
```

图 4　Ruby 编写的阶乘程序

在 Lisp 中，最重要的数据类型是表（List），甚至 Lisp 这个名字本身也是从 List Processor 而来的。一个表是由被称为单元（Cell）的数据连接起来所构成的（图 5）。一个单元包含两个值，一个叫做 car，另一个叫做 cdr。它们的值可以是对其他单元的引用，或者是被称为原子（Atom）的非单元值。例如，数值、字符串、符号等，都属于原子。

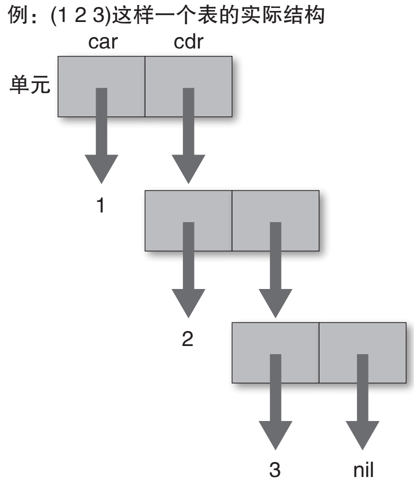


图 5　Lisp 的表

S 表达式是用来描述这种表的记法，它遵循下述规则（语法）。首先，单元是用点对（Dotted pair）来描述的。例如，car 和 cdr 都为数值 1 的单元，要写成下面这样。


```
(1 . 1)
```

其次，cdr 部分如果是一个表，则省略点和括号，也就是说：

```
(1 . (2 . 3))
```

应该写成：

```
(1 2 . 3)
```

然后，如果 cdr 部分为 nil，则省略 cdr 部分。于是：

```
(1 2 3 . nil)
```

应该写成：

```
(1 2 3)
```

S 表达式的基本规则就只有上面这些。只要理解了上述规则，就可以通过“括号的罗列”来想象出实际的表结构。掌握了规则之后再看图 5，应该就能够理解得更加清楚了吧。

那么这里重要的一点是，Lisp 程序是通过 S 表达式来进行表达的。换句话说，Lisp 程序正是通过 Lisp 本身最频繁操作的表的方式来表达的。这意味着程序和数据是完全等同的，在这一点上非常符合元编程的概念，实际上，元编程已经深深融入 Lisp 之中，成为其本质的一部分。

Lisp 程序

Lisp 程序是由形式（Form）排列起来构成的。形式就是 S 表达式，它通过下面的规则来进行求值。

- 符号（Symbol）会被解释为变量，求出该变量所绑定的值。
- 除符号以外的原子，则求出其自身的值。即：整数的话就是该整数本身，字符串的话就是该字符串本身。
- 如果形式为表，则头一个符号为“函数名”，表中剩余的元素为参数①。

在形式中，表示函数名的部分，实际上还分为函数、特殊形式和宏三种类型，它们各自的行为都有所区别。函数相当于 C 语言中的函数，或者 Ruby 中的方法，在将参数求值后，函数就会被调用。特殊形式则相当于其他语言中的控制结构，这些结构是无法通过函数来表达的。例如，Lisp 中用于赋值的 setq 特殊形式，写法如下：

```
(setq a 128)
```

假设 setq 是一个函数，那么 a 作为其参数会被求值，而不会对变量 a 进行赋值。setq 并不会对 a 进行求值，而是将其作为变量名来对待，这是 Lisp 语言中直接设定好的规则，像这样拥有特殊待遇的形式就被称为特殊形式。除了 setq 以外，特殊形式还有用于条件分支的 if 和用于定义局部变量的 let。

宏

对 Lisp 的介绍篇幅比预想的要长。其实，我真正想要介绍的就是这个“宏”（Macro）。Lisp 中的宏，可以在对表达式求值时，通过对构成程序的表进行操作，从而改写程序本身。首先，我们来将它和函数做个比较。

首先，我们来看看对参数进行平方计算的函数 square（图 6 上），以及将参数进行平方计算的宏 square2（图 6 下）的定义。看出区别了吗？

```
(defun square (x)
   (* x x))

(defmacro square2 (x)
  (list '* x x))
```

图 6　函数定义和宏定义

在函数定义中使用了 defun（def function 的缩写），而在宏定义中则用的是 defmacro，这是一点区别。另外，宏所返回的不是

① 这是 CommonLisp 等被称为 Lisp-2 系列的 Lisp 中的行为，Scheme 等属于 Lisp-1 系列的语言中，行为是有区别的。（原书注）

求值的结果，而是以表的形式返回要在宏被调用的地方嵌入的表达式。例如，如果要对：

```
(square2 2)
```

进行求值的话，Lisp 会找到 square2，发现这是一个宏，首先，它会用 2 作为参数，对 square2 本身进行求值。list 是将作为参数传递的值以表的形式返回的函数。

```
(list '* x x) ;; => (* 2 2)
```

然后，将这个结果嵌入到调用 square2 的地方，再进行实际的求值。

虽说就 square 和 square2 来说（如果参数没有副作用的话），两种方法基本上没什么区别，但通过使用获取参数、加工、然后再嵌入的技术，只要是遵循 S 表达式的语法，其可能性就几乎是无限的。无论是创建新的控制结构，还是在 Lisp 中创建其他的语言（内部 DSL）都十分得心应手。

由宏所实现的这种“只要在 S 表达式范围内便无所不能”的性质，是 Lisp 的重要特性之一。实际上，包括 CommonLisp 中用于函数定义的 defun 在内，其语言设计规格中有相当一部分就是通过宏来实现的。

那么，我们来想想看有没有只有通过宏才能实现的例子呢？图 7 的程序是将指定变量内容加 1 的宏 inc。

```
(defmacro inc (var)
   (list 'setq var (list' 1+ var)))
```

图 7　inc 宏

“将变量的内容加 1”这样的操作，由于包含赋值操作，用一般的函数是无法实现的。但是，使用宏就可以很容易地实现这样的扩展操作。inc 宏实际使用的例子如图 8 的 (a) 部分所示。

```
;;; (a) inc宏的调用
(setq a 41)  ;; 变量a初始化
(inc a)      ;; a的值变为42

;;; (b) 查看inc宏的实体
;;; 用macroexpand函数可以查看宏的展开结果
(macroexpand '(inc a))
;;; => (setq a (1+ a))
```

图 8　inc 宏的使用

宏的展开结果可以用 macroexpand 函数来查看。图 8 的 (b) 部分中，我们就用 macroexpand 函数来查看了宏的展开结果，它的展开结果是一个 setq 赋值语句。我们的这个宏非常简单，但如果是复杂的宏，便很难想象出其展开结果会是什么样子，因此 macroexpand 函数对于宏的调试是非常有效的。

宏的功与过

正如刚才讲过的，Lisp 的宏是非常强大的。要展现宏的强大，还有一个例子，就是 CLOS（Common Lisp Object System）。

在 CommonLisp 第一版中，并没有提供面向对象的功能。在第二版中则在其规格中默认包含了这个名为 CLOS 的面向对象功能，这个功能的实现，是通过宏等手段，仅由 CommonLisp 自身完成的。CommonLisp（及其宏）实在是太强大了，在语言本身没有进行增强的情况下，就可以定义出面向对象功能。而之所以默认包含在语言规格中，只是为了消除因为实现方法不同而产生的不安定因素，将这一实现方法用严密的格式写成了文档而已[①]。

而且，CLOS 并不是只是一个做出来玩玩的玩具，而是一个真正意义上的，拥有大量丰富功能的复杂的面向对象系统，实现了同时代的其他语言到现在为止都未能实现的功能[②]。

例如，Ruby 虽然是一种非常灵活的动态语言，但它的面向对象功能也是用内嵌的方式来实现的，靠语言本身的能力来实现面向对象的功能是做不到的。而这样的功能，Lisp 却仅仅通过语言本身的能力定义了出来，不得不说 Lisp 和它的宏简直强大到令人发指。

既然宏如此强大，那为什么 Ruby 等其他语言中没有采用 Lisp 风格的宏呢？

其中一个原因是语法的问题。Lisp 宏的强大力量，源于程序和数据采用相同结构这一点。然而与此同时，Lisp 程序中充满了括号，并不是“一般的程序员”所熟悉和习惯的语法。

作为一个语言设计者，自己的语言是否要采用 S 表达式，是一个重大的决策。为了强大的宏而牺牲语法的易读和易懂性，做出这样的判断是十分困难的。

在没有采用 S 表达式的语言中，也有一些提供了宏功能。例如 C（和 C++）中的宏是用预处理器（Preprocessor）来实现的，不过，这种方式只能做简单的字符串替换，无法编写复杂的宏。以前，美国苹果公司开发过一种叫做 Dylan[③]的语言，采用了和 Algol 类似的（比较一般的）语法，但也对宏的实现做出了尝试，不过由于诸多原因，它还没有普及就夭折了。

另一个难点在于，如果采用了宏，程序的解读就会变得困难。

宏的优点在于，包括控制结构的定义在内，只要在 S 表达式语法的范围内就可以实现任何功能，但这些功能也仅限于增强语言的描述能力和提供内部 DSL（特定领域语言）而已，此外

① 作者在给译者的电子邮件中，对这一内容做出如下补充：由于 CLOS 是只用 CommonLisp 内置功能（如宏等）来实现的，因此实际上任何人都可以用不同的方法实现类似的功能，为了让 CLOS 能够跨解释器通用，因此在文档中对其实现方式进行了标准化。当然，并不是所有的 CommonLisp 解释器中的 CLOS 都是仅通过宏来实现的，出于速度方面的优化等考虑，将 CLOS 直接嵌入到解释器中的做法也很常见。

② CLOS 定义于 1988 年。（原书注）

③ Dylan（名称来自 DYnamic LANguage 的缩写）是一种多范式跨平台编程语言，由苹果公司于 20 世纪 90 年代初开始开发，后来项目被终止，只发布了一个技术版本。后来 Harlequin 公司和卡内基梅隆大学的团队分别发布了 Dylan 的 Windows 和 Unix 版本，目前由开源社区 Open Dylan 运营维护。

并没有什么更高级的用法了。不过，反过来说，这也意味着如果不具备宏所提供的新语法的相关知识，就很难把握程序的含义。如果你让 Lisp 高手谈谈关于宏的话题，他们大概会异口同声地说："宏千万不能多用，只能在关键时刻用一下。"说起来，元编程本身也差不多是这样一个趋势吧。

不过，作为 Ruby 语言的设计者，依我看，宏的使用目的中很大的一部分，主观判断大约有六七成的情况，其实都可以通过 Ruby 的代码块来实现。我的看法是，从这个角度来说，在 Ruby 中提供宏功能，实际上是弊大于利的。然而，追求更强大的功能是程序员的天性，我也经常听到希望 Ruby 增加宏功能的意见，据说甚至有人通过修改 Ruby 的解释器，已经把宏功能给搞出来了。唔……

元编程的可能性与危险性

在 Ruby 和 Lisp 这样的语言中，由于程序本身的信息是可以被访问的，因此在程序运行过程中也可以对程序本身进行操作，这就是元编程。使用元编程技术，可以实现通常情况下无法实现的操作。例如，Ruby on Rails 的数据库适配器 ActiveRecord 可以读取数据库结构，通过元编程技术在运行时添加用于访问数据库记录的方法。这样一来，即便数据库结构发生变化，在软件一侧也没有必要做出任何修改。

再举一个例子，我们来看看 Builder 这个库。Builder 是用于生成标记语言（Mark-up language）代码的库，应用示例如图 9 所示。

```
require 'builder'

builder = Builder::XmlMarkup.new
xml = builder.person {|b|
  b.name("Jim")
  b.phone("555-1234")
}
#=> <person><name>Jim</name><phone>555-1234</phone></person>
```

图 9　Builder 库的应用

在图 9 的示例中，person 和 name、phone 等标签是作为方法来调用的，但这些方法并不是由 Builder 库所定义的。由于 XML（Extensible Markup Language，可扩展标记语言）中并没有事先规定要使用哪些标签，因此在库中对标签进行预先定义是不可能的。于是，在 Builder 库中，是通过元编程技术，用钩子（Hook）截获要调用的方法，来生成所需的标签的。

无论是 ActiveRecord 的示例，还是 Builder 的示例，都通过元编程技术对无法预先确定的操

作进行了应对，这样一来，未来的可能性就不会被禁锢，体现了语言的灵活性。我认为，这种灵活性正是元编程最大的力量。

另一方面，元编程技术如果用得太多，编写出来的程序就很难一下子看明白。例如，在 Builder 库的源代码中，怎么看也找不到定义 person 方法的部分，如果没有元编程知识的话，要理解源代码就很困难。和宏一样，元编程的使用也需要掌握充分的知识，并遵守用量和用法。

小结

读过《Ruby 元编程》一书之后，印象最深的是下面这段。

> 根本没有什么元编程，只有编程而已。（中略）这句话让弟子茅塞顿开。

的确如此。程序是由数据结构和算法构成的，然而，如果环境允许程序本身作为数据结构来操作的话，那么元编程也就和面向一般数据结构的一般操作没什么两样了。作为像 Lisp 和 Ruby 这样允许对程序结构进行访问的语言来说，所谓元编程，实际上并不是什么特殊的东西，而只不过是日常编程的一部分罢了。

2.4 内存管理

在现实世界中总有这样那样的局限和制约，但计算机将人类从那些局限中解放出来——至少是在试图努力实现这个目标。然而，计算机也是现实世界存在的一部分，当然其本身也会受到制约。因此，计算机只是提供了一种幻觉，让我们人类以为自己已经从这些制约中解放出来了。

看似无限的内存

我们来具体讲讲吧。比如说，内存。大家的电脑上面装了多大的内存呢？最近的电脑内存大多都有几个 GB 吧，我手上的笔记本电脑内存有 8GB。我上高中的时候，电脑只有 32KB 的 RAM，8GB 的容量相当于其 25 万倍了。也就是说，在这 30 年中，一般人可以获得的内存容量是 30 年前的 25 万倍。25 万倍……

不过，无论内存容量有多大，总归不是无限的。实际上，伴随着内存容量的增加，软件的内存开销也在以同样的速率增加着。因此，最近的计算机系统会通过“双重”幻觉，让我们以为内存容量是无限的。

第一重幻觉是垃圾回收（GC）机制。关于这一点我们稍后会详细讲解。

第二重幻觉是操作系统提供的虚拟内存。由于硬盘的容量要远远大于内存（RAM），虚拟内存正是利用这一点，在内存容量不足时将不经常被访问的内存空间中的数据写入硬盘，以增加“账面上”可用内存容量的手段。现在，虽说内存容量已经增加了很多，但也不过是区区几个 GB 而已。相对的，即便是笔记本电脑上的硬盘，也已经有几百 GB 的容量，超过 1TB（1000GB）的也开始出现了。虚拟内存也就是利用了这样的容量差异。

书桌上的文件摊满了，也就没地方放新的文件了。所谓虚拟内存，就好比是将书桌上比较老的文件先暂时收到抽屉里，用空出来的地方来摊开新的文件。

不过，如果在书桌和抽屉之间频繁进行文件的交换，工作效率肯定会下降。如果每次要看一份文件都要先收拾书桌再到抽屉里面拿的话，那工作根本就无法进行了。虚拟内存也有同样的缺点。硬盘的容量比内存大，但相对的，速度却非常缓慢，如果和硬盘之间的数据交换过于

频繁，处理速度就会下降，表面上看起来就像卡住了一样，这种现象被称为抖动（Thrushing）。应该有很多人有过计算机停止响应的经历，而造成死机的主要原因之一就是抖动。

GC 的三种基本方式

好了，下面我们来讲讲 GC（Garbage Collection）。在 Java 和 Ruby 这样的语言中，程序在运行时会创建很多对象。从编程语言的角度来看，它们是对象；但从计算机的角度来看，它们也就是一些装有数据的内存空间而已。

在 C 和 C++ 这样的语言中，这些内存空间是由人手动进行管理的。当需要内存空间时，要请求操作系统进行分配，不需要的时候要返还给操作系统。然而，正是“不再需要”这一点，带来了各种各样的麻烦。

因为“不再需要”而返还给操作系统的内存空间，会被操作系统重新利用，如果不小心访问了这些空间的话，里面的数据会被改写，这会造成程序的异常行为，甚至是崩溃。反过来说，如果认为某些内存空间“可能还要用到”而不还给操作系统，或者是用完了却忘记返还，这些无法访问的空间就会一直保留下来，造成内存的白白浪费，最终引发性能下降和产生抖动。从结果来看，让人来管理大量分配的内存空间，是非常困难的。

将内存管理，尤其是内存空间的释放实现自动化，这就是 GC。GC 其实是个古老的技术，从 20 世纪 60 年代就开始研究，还发表了不少论文。这项技术在大学实验室级别的地方已经应用了很长时间，但是可以说，从 20 世纪 90 年代 Java 出现之后，一般的程序员才有缘接触到它。在此之前，这项技术还只是少数人的专利。

术语定义

在讲解 GC 技术之前，我们先来定义两个即将用到的术语。

1. 垃圾

所谓垃圾（Garbage），就是需要回收的对象。作为编写程序的人，是可以做出“这个对象已经不再需要了”这样的判断，但计算机是做不到的。因此，如果程序（通过某个变量等等）可能会直接或间接地引用一个对象，那么这个对象就被视为“存活”；与之相反，已经引用不到的对象被视为“死亡”。将这些“死亡”对象找出来，然后作为垃圾进行回收，这就是 GC 的本质。

2. 根

所谓根（Root），就是判断对象是否可被引用的起始点。至于哪里才是根，不同的语言和编译器都有不同的规定，但基本上是将变量和运行栈空间作为根。

好了，用上面这两个术语，我们来讲一讲主要的 GC 算法。

标记清除方式

标记清除（Mark and Sweep）是最早开发出的 GC 算法（1960 年）。它的原理非常简单，首先从根开始将可能被引用的对象用递归的方式进行标记，然后将没有标记到的对象作为垃圾进行回收。

图 1 显示了标记清除算法的大致原理。

图 1 中的 (1) 部分显示了随着程序的运行而分配出一些对象的状态，一个对象可以对其他的对象进行引用。

图中 (2) 部分中，GC 开始执行，从根开始对可能被引用的对象打上“标记”。大多数情况下，这种标记是通过对象内部的标志（Flag）来实现的。于是，被标记的对象我们把它们涂黑。

图中 (3) 部分中，被标记的对象所能够引用的对象也被打上标记。重复这一步骤的话，就可以将从根开始可能被间接引用到的对象全部打上标记。到此为止的操作，称为标记阶段（Mark phase）。标记阶段完成时，被标记的对象就被视为“存活”对象。

图 1 中的 (4) 部分中，将全部对象按顺序扫描一遍，将没有被标记的对象进行回收。这一操作被称为清除阶段（Sweep phase）。在扫描的同时，还需要将存活对象的标记清除掉，以便为下一次 GC 操作做好准备。

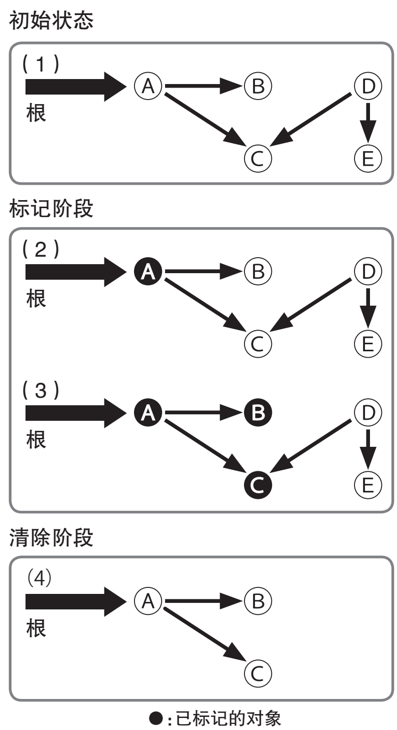


图 1　标记清除算法

标记清除算法的处理时间，是和存活对象数与对象总数的总和相关的。

作为标记清除的变形，还有一种叫做标记压缩（Mark and Compact）的算法，它不是将被标记的对象清除，而是将它们不断压缩。

复制收集方式

标记清除算法有一个缺点，就是在分配了大量对象，并且其中只有一小部分存活的情况下，所消耗的时间会大大超过必要的值，这是因为在清除阶段还需要对大量死亡对象进行扫描。

复制收集（Copy and Collection）则试图克服这一缺点。在这种算法中，会将从根开始被引用的对象复制到另外的空间中，然后，再将复制的对象所能够引用的对象用递归的方式不断复制下去。

图 2 的 (1) 部分是 GC 开始前的内存状态，这和图 1 的 (1) 部分是一样的。图 2 的 (2) 部分中，在旧对象所在的“旧空间”以外,再准备出一块“新空间”,并将可能从根被引用的对象复制到新空间中。图中 (3) 部分中，从已经复制的对象开始，再将可以被引用的对象像一串糖葫芦一样复制到新空间中。复制完成之后，“死亡”对象就被留在了旧空间中。图中 (4) 部分中，将旧空间废弃掉，就可以将死亡对象所占用的空间一口气全部释放出来，而没有必要再次扫描每个对象。下次 GC 的时候，现在的新空间也就变成了将来的旧空间。

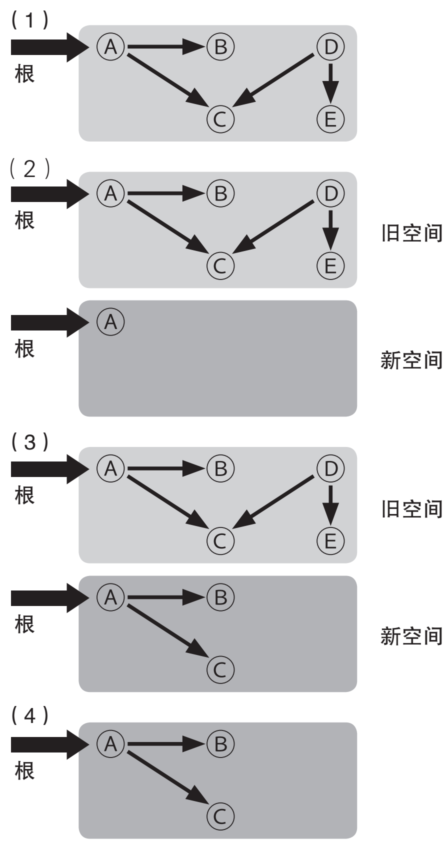


图 2　复制收集算法

通过图 2 我们可以发现，复制收集方式中，只存在相当于标记清除方式中的标记阶段。由于清除阶段中需要对现存的所有对象进行扫描,在存在大量对象,且其中大部分都即将死亡的情况下，全部扫描一遍的开销实在是不小。

而在复制收集方式中，就不存在这样的开销。但是，和标记相比，将对象复制一份所需要的开销则比较大，因此在“存活”对象比例较高的情况下，反而会比较不利。

这种算法的另一个好处是它具有局部性（Locality）。在复制收集过程中，会按照对象被引用的顺序将对象复制到新空间中。于是，关系较近的对象被放在距离较近的内存空间中的可能性会提高，这被称为局部性。局部性高的情况下，内存缓存会更容易有效运作，程序的运行性能也能够得到提高。

引用计数方式

引用计数（Reference Count）方式是 GC 算法中最简单也最容易实现的一种，它和标记清除方式差不多是在同一时间发明出来的。

它的基本原理是，在每个对象中保存该对象的引用计数，当引用发生增减时对计数进行更新。

引用计数的增减，一般发生在变量赋值、对象内容更新、函数结束（局部变量不再被引用）等时间点。当一个对象的引用计数变为 0 时，则说明它将来不会再被引用，因此可以释放相应的内存空间。

图 3 的 (1) 部分中，所有对象中都保存着自己被多少个其他对象进行引用的数量（引用计数），图中每个对象右上角的数字就是引用计数。

图中 (2) 部分中，当对象引用发生变化时，引用计数也跟着变化。在这里，由对象 B 到对象 D 的引用失效了，于是对象 D 的引用计数变为 0。由于对象 D 的引用计数为 0，因此由对象 D 到对象 C 和 E 的引用数也分别相应减少。结果，对象 E 的引用计数也变为 0，于是对象 E 也被释放掉了。

图 3 的 (3) 部分中，引用计数变为 0 的对象被释放，“存活”对象则保留了下来。大家应该注意到，在整个 GC 处理过程中，并不需要对所有对象进行扫描。

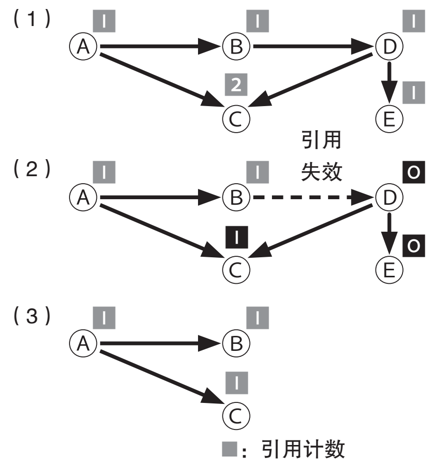


图 3 引用计数算法

实现容易是引用计数算法最大的优点。标记清除和复制收集这些 GC 机制在实现上都有一定难度；而引用计数方式的话，凡是有些年头的 C++ 程序员（包括我在内），应该都曾经实现过类似的机制，可以说这种算法是相当具有普遍性的。

除此之外，当对象不再被引用的瞬间就会被释放，这也是一个优点。其他的 GC 机制中，要预测一个对象何时会被释放是很困难的，而在引用计数方式中则是立即被释放的。而且，由于释放操作是针对每个对象个别执行的，因此和其他算法相比，由 GC 而产生的中断时间（Pause time）就比较短，这也是一个优点。

引用计数方式的缺点

另一方面，这种方式也有缺点。引用计数最大的缺点，就是无法释放循环引用的对象。图 4 中，

A、B、C 三个对象没有被其他对象引用，而是互相之间循环引用，因此它们的引用计数永远不会为 0，结果这些对象就永远不会被释放。

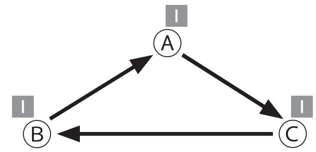


图 4 无法回收，循环引用

引用计数的第二个缺点，就是必须在引用发生增减时对引用计数做出正确的增减，而如果漏掉了某个增减的话，就会引发很难找到原因的内存错误。引用数忘了增加的话，会对不恰当的对象进行释放；而引用数忘了减少的话，对象会一直残留在内存中，从而导致内存泄漏。如果语言编译器本身对引用计数进行管理的话还好，否则，如果是手动管理引用计数的话，那将成为孕育 bug 的温床。

最后一个缺点就是，引用计数管理并不适合并行处理。如果多个线程同时对引用计数进行增减的话，引用计数的值就可能会产生不一致的问题（结果则会导致内存错误）。为了避免这种情况的发生，对引用计数的操作必须采用独占的方式来进行。如果引用操作频繁发生，每次都要使用加锁等并发控制机制的话，其开销也是不可小觑的。

综上所述，引用计数方式的原理和实现虽然简单，但缺点也很多，因此最近基本上不再使用了。现在，依然采用引用计数方式的语言主要有 Perl 和 Python，但它们为了避免循环引用的问题，都配合使用了其他的 GC 机制。这些语言中，GC 基本上是通过引用计数方式来进行的，但偶尔也会用其他的算法来执行 GC，这样就可以将引用计数方式无法回收的那些对象处理掉。

进一步改良的应用方式

GC 的基本算法，大体上都逃不出上述三种方式以及它们的衍生品。现在，通过对这三种方式进行融合，出现了一些更加高级的方式。这里，我们介绍一下其中最有代表性的三种，即分代回收、增量回收和并行回收。有些情况下，也可以对这些方法中的几种进行组合使用。

分代回收

首先，我们来讲讲高级 GC 技术中最重要的一种，即分代回收（Generational GC）。

由于 GC 和程序处理的本质是无关的，因此它所消耗的时间越短越好。分代回收的目的，正是为了在程序运行期间，将 GC 所消耗的时间尽量缩短。

分代回收的基本思路，是利用了一般性程序所具备的性质，即大部分对象都会在短时间内成为垃圾，而经过一定时间依然存活的对象往往拥有较长的寿命。如果寿命长的对象更容易存

活下来，寿命短的对象则会被很快废弃，那么到底怎样做才能让 GC 变得更加高效呢？如果对分配不久，诞生时间较短的“年轻”对象进行重点扫描，应该就可以更有效地回收大部分垃圾。

在分代回收中，对象按照生成时间进行分代，刚刚生成不久的年轻对象划为新生代（Young generation），而存活了较长时间的对象划为老生代（Old generation）。根据具体实现方式的不同，可能还会划分更多的代，在这里为了讲解方便，我们就先限定为两代。如果上述关于对象寿命的假说成立的话，那么只要仅仅扫描新生代对象，就可以回收掉废弃对象中的很大一部分。

像这种只扫描新生代对象的回收操作，被称为小回收（Minor GC）。小回收的具体回收步骤如下。

首先从根开始一次常规扫描，找到“存活”对象。这个步骤采用标记清除或者是复制收集算法都可以，不过大多数分代回收的实现都采用了复制收集算法。需要注意的是，在扫描的过程中，如果遇到属于老生代的对象，则不对该对象继续进行递归扫描。这样一来，需要扫描的对象数量就会大幅减少。

然后，将第一次扫描后残留下来的对象划分到老生代。具体来说，如果是用复制收集算法的话，只要将复制目标空间设置为老生代就可以了；而用标记清除算法的话，则大多采用在对象上设置某种标志的方式。

对来自老生代的引用进行记录

这个时候，问题出现了，从老生代对象对新生代对象的引用怎么办呢？如果只扫描新生代区域的话，那么从老生代对新生代的引用就不会被检测到。这样一来，如果一个年轻的对象只有来自老生代对象的引用，就会被误认为已经“死亡”了。因此，在分代回收中，会对对象的更新进行监视，将从老生代对新生代的引用，记录在一个叫做记录集（remembered set）的表中（图 5）。在执行小回收的过程中，这个记录集也作为一个根来对待。

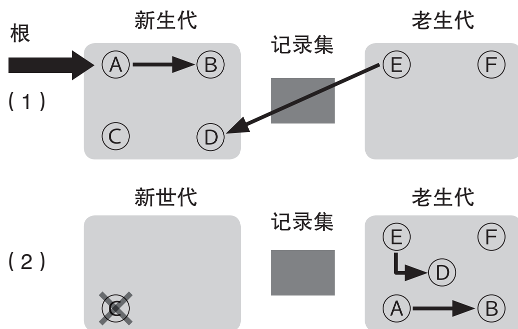


图 5 分代回收方式中的小回收

从任何地方都没有进行引用的老生代中的 F 对象，会通过大回收操作进行回收。

要让分代回收正确工作，必须使记录集的内容保持更新。为此，在老生代到新生代的引用产生的瞬间，就必须对该引用进行记录，而负责执行这个操作的子程序，需要被嵌入到所有涉及对象更新操作的地方。

这个负责记录引用的子程序是这样工作的。设有两个对象:A 和 B,当对 A 的内容进行改写,并加入对 B 的引用时，如果① A 属于老生代对象，② B 属于新生代对象，则将该引用添加到记录集中。

这种检查程序需要对所有涉及修改对象内容的地方进行保护，因此被称为写屏障（Write barrier）。写屏障不仅用于分代回收，同时也用在很多其他的 GC 算法中。

虽说老生代区域中的对象一般来说寿命都比较长，但也决不是“不老不死”的。随着程序的运行，老生代区域中的“死亡”对象也在不断增加。为了避免这些死亡的老生代对象白白占用内存空间，偶尔需要对包括老生代区域在内的全部区域进行一次扫描回收。像这样以全部区域为对象的 GC 操作被称为完全回收（Full GC）或者大回收（Major GC）。

分代回收通过减少 GC 中扫描的对象数量，达到缩短 GC 带来的平均中断时间的效果。不过由于还是需要进行大回收，因此最大中断时间并没有得到什么改善。从吞吐量来看，在对象寿命假说能够成立的程序中，由于扫描对象数量的减少，可以达到非常不错的成绩。但是，其性能会被程序行为、分代数量、大回收触发条件等因素大幅度左右。

增量回收

在对实时性要求很高的程序中，比起缩短 GC 的平均中断时间，往往更重视缩短 GC 的最大中断时间。例如，在机器人的姿势控制程序中，如果因为 GC 而让控制程序中断了 0.1 秒，机器人可能就摔倒了。或者，如果车辆制动控制程序因为 GC 而延迟响应的话，后果也是不堪设想的。

在这些对实时性要求很高的程序中，必须能够对 GC 所产生的中断时间做出预测。例如，可以将“最多只能中断 10 毫秒”作为附加条件。

在一般的 GC 算法中，作出这样的保证是不可能的，因为 GC 产生的中断时间与对象的数量和状态有关。因此，为了维持程序的实时性，不等到 GC 全部完成，而是将 GC 操作细分成多个部分逐一执行。这种方式被称为增量回收（Incremental GC）。

在增量回收中，由于 GC 过程是渐进的，在回收过程中程序本身会继续运行，对象之间的引用关系也可能会发生变化。如果已经完成扫描和标记的对象被修改，对新的对象产生了引用，这个新对象就不会被标记，明明是“存活”对象却被回收掉了。

在增量回收中为了避免这样的问题，和分代回收一样也采用了写屏障。当已经被标记的对象的引用关系发生变化时，通过写屏障会将新被引用的对象作为扫描的起始点记录下来。

由于增量回收的过程是分步渐进式的，可以将中断时间控制在一定长度之内。另一方面，由于中断操作需要消耗一定的时间，GC 所消耗的总时间就会相应增加，正所谓有得必有失。

并行回收

最近的计算机中，一块芯片上搭载多个 CPU 核心的多核处理器已经逐渐普及。不仅是服务器，就连个人桌面电脑中，多核 CPU 也已经成了家常便饭。例如美国英特尔公司的 Core i7 就拥有 6 核 12 个线程。

在这样的环境中，就需要通过利用多线程来充分发挥多 CPU 的性能。并行回收正是通过最大限度利用多 CPU 的处理能力来进行 GC 操作的一种方式。

并行回收的基本原理是，是在原有的程序运行的同时进行 GC 操作，这一点和增量回收是相似的。不过，相对于在一个 CPU 上进行 GC 任务分割的增量回收来说，并行回收可以利用多 CPU 的性能，尽可能让这些 GC 任务并行（同时）进行。由于软件运行和 GC 操作是同时进行的，因此就会遇到和增量回收相同的问题。为了解决这个问题，并行回收也需要用写屏障来对当前的状态信息保持更新。不过，让 GC 操作完全并行，而一点都不影响原有程序的运行，是做不到的。因此在 GC 操作的某些特定阶段，还是需要暂停原有程序的运行。

在多核化快速发展的现在，并行回收也成了一个非常重要的话题，它的算法也在不断进行改善。在硬件系统的支持下，无需中断原有程序的完全并行回收器也已经呼之欲出。今后，这个领域相当值得期待。

GC 大统一理论

像标记清除和复制收集这样，从根开始进行扫描以判断对象生死的算法，被称为跟踪回收（Tracing GC）。相对的，引用计数算法则是当对象之间的引用关系发生变化时，通过对引用计数进行更新来判定对象生死的。

美国 IBM 公司沃森研究中心的 David F. Bacon 等人，于 2004 年发表了一篇题为“垃圾回收的统一理论”（A Unified Theory of Garbage Collection）的论文，文中阐述了一种理论，即：任何一种 GC 算法，都是跟踪回收和引用计数回收两种思路的组合。两者的关系正如“物质”和“反物质”一样，是相互对立的。对其中一方进行改善的技术之中，必然存在对另一方进行改善的技术，而其结果只是两者的组合而已。

例如，用于改善分代回收和增量回收等跟踪回收算法的写屏障机制，从对引用状态变化进行记录这个角度来看，就是吸收了引用计数回收的思路。相对的，引用计数算法也吸收了分代回收算法的思路而进行了一些改进，如来自局部变量的引用变化不改变引用计数等。

Unified Theory 来源于物理学中的大统一理论（Grand Unified Theory，简称 GUT）一词。正如试图统一解释自然界中四种基本作用力的大统一理论一样，这个试图统一解释跟踪回收和引用计数回收的理论，也就被命名为 GC 大统一理论了。

2.5 异常处理

不知道大家有没有听说过“正常化偏见”（normalcy bias）这个词。所谓正常化偏见指的是人们的一种心理倾向，对于一些偶然发生的情况，一旦发生了便会不自觉地忽略其危害。

在之前发生的大地震[①]中，虽然发布了海啸预警，但据说还是有很多人，由于觉得“这次也没什么大不了的”、“海啸不会袭击这里的”而不幸遇难。我认为，这并不是说那些人很愚蠢，而是说明人类是很容易受到“正常化偏见”这种心理倾向的影响的。如果遇到同样的状况，换做你和我的话，很可能也会做出同样错误的判断。

“一定没问题的”

程序员也是人，同样无法逃脱正常化偏见的影响。对于程序运行中所发生的异常情况，总是会觉得“这种情况一般不会出现的”、“所以不解决也没关系”。例如，大家可能会这样想：“配置文件肯定会被安装进去的，因此不必考虑配置文件不存在的情况”，“网络通信中丢包之类的问题 TCP 层会帮忙搞定的，因此应该不用考虑通信失败的情况”。总是把情况往好的方面设想，这样的心理在程序员中很常见。

然而，正如墨菲定律[②]所说的，即便是极少会发生的情况，只要有发生的可能性，早晚是会发生的。说不定就有人不小心把配置文件手动删除了，也说不定就在网络通信过程中路由器断电了，计划永远赶不上变化。一旦发生异常情况，就该怪自己平时没做好应对了。“哎呀，早知道当初就该好好应对的”，现在才意识到这一点，也只能是马后炮了。

软件开发的历史，就是和 bug 斗争的历史。最早的 bug 是由于一只臭虫（bug）卡在组成计算机的继电器中所引发的。在开发软件的过程中，几乎不会有人想到会有虫子卡在电路里面吧，这真是一个意外。不过，在软件开发中，还是必须对各种事态都做出预计才行。

① 这里指的是 2011 年 3 月 11 日发生的日本东北地方太平洋近海地震，震级达到 9.0 级。

② 墨菲定律（Murphy’s Law），原来的表述是“凡是可能出错的事都会出错”（Anything that can go wrong will go wrong），意思是说，任何一个事件只要发生的概率大于零，就不能假设它不会发生。

用特殊返回值表示错误

那么，作为例题，我们来举一个非常简单的打开文件操作的例子。在 C 语言中，将文件以读方式打开的程序如图 1 所示。

```
#include <stdio.h>

int
main()
{
 FILE *f = fopen("/path/to/file",
"r");

 if (f == NULL) {
   puts("file open failed");
 }
 else {
   puts("file open succeeded");
 }
}
```

图 1　C 语言中的文件打开操作

C 语言的打开文件函数 fopen，会将位于指定路径的文件以指定的模式（读 / 写 / 追加）打开。打开成功时，返回指向 FILE 结构体的指针；打开失败时，则返回 NULL。

让 fopen 返回 NULL 的原因有很多，全部列举出来实在太难了。下面举几个有代表性的例子：

- ❑ 文件不存在
- ❑ 没有权限访问该文件
- ❑ 该进程中已打开的文件数量太多
- ❑ 指定的路径不是一个文件而是一个目录
- ❑ 内核内存不足
- ❑ 磁盘已满
- ❑ 指定了非法的路径地址

上面这些只不过是失败原因的一部分而已，感觉很头大吧。

在 C 语言中，表示错误的主要方式是通过“特殊返回值”。大多数情况下，和 fopen 一样，通过返回 NULL 来表示错误。

容易忽略错误处理

使用特殊返回值这个方法不需要编程语言的支持，是一种非常简便的方法，但它有两个重大的缺点。

第一，由于对错误的检测不是强制进行的，可能会出现没有注意到发生了错误而继续运行程序的情况。如果没有注意到文件打开失败，依然去访问 FILE 结构体的话，整个程序就会出错崩溃。仅仅因为要打开的文件不存在就崩溃的程序，实在是太差劲了。

对于文件不存在这种比较常见的状况，一般来说大概不会疏于应对。不过，发生概率比较低的意外情况却很容易被忽略。例如，分配内存的函数 malloc，在内存不足时会返回 NULL 以表示错误，这一点在文档上写得清清楚楚，却还是有很多程序没有做出应对。如果写一个总是返回 NULL 的 malloc 函数连接上去的话，就会惊奇地发现居然有那么多程序根本就不检查 malloc 的返回值。

第二，原本的程序容易被错误处理埋没。错误处理是对意外性的异常事态所做的应对，并不是我们本来想做的事。然而，正如之前讲过的，我们又不能忽略错误的存在，于是本来只是配角的错误处理部分就会在程序中喧宾夺主。

我想执行的是一系列简单的操作：打开文件，从文件中逐行读取内容，加工之后输出到标准输出设备（stdout）。而实际的代码却变成了十分繁琐的内容：打开文件……打开了吗？没打开的话显示错误信息然后程序结束；读取 1 行内容……读取成功了吗？没成功的话，如果到了文件末尾则程序结束，如果没到文件末尾，则忽略该行；将读取的内容进行加工，然后输出结果；加工过程中如果发生错误，别忘了对错误进行处理……

你有没有觉得太麻烦了？这种感觉太正常了。不过，受过良好训练的 C 语言程序员则不会有任何怨言，因为他们多年以来一直都在重复着这样的辛苦工作。

Ruby 中的异常处理

那么，对于这样的“错误地狱”，编程语言方面又提供了怎样的支持呢？

正如上面所总结的，其实问题点有两个：没有检查错误就继续运行，错误处理将原本的程序埋没。

于是，在比较新的语言中，采用了称为异常（exception）的机制，以减轻错误处理的负担。一旦发生意外情况，程序中就会产生异常，并同时中断程序运行，回溯到当前过程的调用者。经过逐级回溯，到达程序顶层之后，输出错误信息，并停止程序的运行。不过，如果明确声明了“在这里捕获异常”的话，异常就会被捕获，并进行错误处理。

```
f = open("/path/to/file", "r")
puts("file open succeeded")
```

图 2　Ruby 中的文件打开操作

图 2 是将图 1 程序所执行的操作，用 Ruby 来编写的程序。当调用用来打开文件的 open 方法时，会返回一个 File 对象。如果发生错误，open 的执行就会中断。在这里我们没有对捕获异常进行声明，因此产生的异常就不会被捕获，程序会显示错误信息，并终止运行。异常事态发生时

的运行终止和错误信息的输出都是自动完成的，这样一来程序便可以集中完成它的本职工作。

在 Ruby 中，对异常的捕捉使用 begin 语句来完成。begin 所包围的代码中如果产生了异常，则会执行 rescue 部分的代码（图 3）。

```
begin
 f = open("/path/to/file", "r")
 puts("file open succeeded")
rescue
 puts("file open failed")
end
```

图 3　Ruby 中的异常处理

由于有了这样的异常处理机制，在 C 语言流派中的显式错误检查所具有的那两个问题得到了一定的缓解。也就是说，当意外状况发生时，通过自动中断程序运行的方式，避免了每进行一步操作都要显式地检查错误，从而也就避免了程序中充满错误检查代码的问题。

不过，当产生异常时也不能总是让程序结束运行，当显式声明需要进行错误处理时，可以恢复产生的错误，并让程序继续运行。

产生异常

下面我们来看看如何人为产生异常。产生异常，可以使用 raise 方法。在 Ruby 中，raise 并不是一个保留字，而是一个方法。raise 方法被调用时，会创建一个用来表示异常的对象，并中断程序运行。在这个过程中，如果存在与异常对象匹配的 rescue 代码，则跳转到该处进行异常处理。

raise 方法的调用有好几种方式，可以根据状况选择合适的调用方式。首先，最基本的方式是仅指定一条错误信息。

```
raise "something bad happens"
```

这条语句会产生一个 RuntimeError 异常。如果不在意异常的类型，只要表达出有错误信息就可以的话，用这种方式是没有问题的。

下面这种方式同时指定了异常类和错误信息。

```
raise TypeError, "wrong type given"
```

这里指定了 Exception 类的一个子类作为异常类。raise 会在内部创建一个指定类的实例，并中断当前程序的运行。第 2 种方式中，还有一个可选的第 3 参数，这个参数可以传递一个数组，用于保存回溯（backtrace，即从哪个函数的第几行进行的调用）信息。

如果要在 rescue 部分中重新产生异常，可以在 raise 方法中指定一个异常对象。

```
raise exc
```

在这种方式中，包含回溯在内的异常信息都被保存在对象中，从而可以将异常抛给位于其上层的代码进行处理。

还有最后一种方式，即可以省略所有的参数，直接调用 raise 方法。如果在 rescue 中用这种方式进行调用的话，会重新产生最近产生过的一个异常。如果在 rescue 外面的话，则会产生一个错误信息为空的 RuntimeError。

更高级的异常处理

用于异常处理的 rescue，会捕获到 begin 所包围的区域中产生的异常，但在这个范围内可能产生的异常往往不止一种。通过在 rescue 后面指定异常的种类（类），就可以针对不同种类的异常分别做出不同的应对（图 4）。更详细的异常信息可以通过在“=>”后面指定变量名来获取。

产生异常时的应对方法，原则上分为两种。一种是中断运行。由于异常产生时会跳转到 rescue，因此可以说中断运行是异常处理的默认方式。

当然，有些情况下，我们并不希望整个程序都停止运行。例如，编辑器要读取一个文件，即便指定文件名不存在，也不能光弹出一条错误信息就退出了吧？这种情况下，应该通过异常处理程序弹出一个警告对话框，然后返回并重新接受用户输入才对。这其实也是中断运行的一个变种。

```
begin
 f = open("/path/to/file", "r")
 puts("file open succeeded")
rescue Errno::ENOENT => e
 puts("file open failed by ENOENT")
rescue ArgumentError => e
 puts("file open failed by
ArgumentError")
end
```

图 4 对多个异常的处理

另一种应对方法，是消除产生异常的原因并重试。为此，Ruby 中有一个 retry 语句，在 rescue 中调用 retry 的话，会跳转回相应的 begin 处重新运行。

图 5 中的程序就是应用 retry 的一个例子。这次我们用 open 方法以写模式打开一个名为 /tmp/foo/file 的文件。然而，/tmp/foo 这个目录不存在的话，就会产生异常。于是在 rescue 中，我们用 mkdir 创建该目录，然后再执行 retry。这样一来，程序会返回 begin 的部分重新运行，这次 open 就可以成功打开文件了。

```
begin
 f = open("/tmp/foo/file", "w")
 puts("file open succeeded")
rescue Errno::ENOENT => e
 puts("file open failed by ENOENT")
 Dir.mkdir("/tmp/foo")
 retry
rescue ArgumentError
 puts("file open failed by
ArgumentError")
end
```

图 5 调用 retry 进行重试

通过 retry 可以在异常处理中实现重试的操作，非常方便。不过它也有一个缺点，那就是如果在 retry 之前没有仔细检查是否对产生异常的条件进行了充分应对的话，就很有可能陷入死循环。

在异常处理完成之后，有时还需要转移到上层的异常处理程序做进一步处理。刚才已经讲过，在 rescue 中直接调用 raise 就可以重新产生异常（图 6）。例如，如果要直接显示顶层错误信息的话，就可以使用这种方式。

```
begin
 # 可能会产生异常的处理
rescue
 # 异常处理程序。输出消息
 puts "exception happened"
 # 重新产生异常
 raise
end
```

图 6　重新产生异常

Ruby 中的后处理保证

rescue 是用来在产生异常的时候进行错误处理的，除此之外，还有一种方式，可以执行一些无论是否产生异常都需要进行的一些清理工作。

以打开文件的操作为例，当处理完成后，无论是正常结束，还是产生了异常，都必须将文件关闭。

在 Ruby 中，使用 open 方法可以保证将打开的文件进行关闭操作（图 7）。如果在调用 open 方法时附加一个代码块，当代码块执行完毕后，就会自动关闭文件。

```
open("/path/to/file", "r") do |f|
 # 对f的处理
end
```

图 7　带代码块的 open

那么，这样的机制如果要自己来实现的话，该如何做呢？

在 Ruby 中，可以使用 ensure。在 begin 部分中如果指定了 ensure，则 begin 部分执行完毕后必定会执行 ensure 部分。这里所说的“执行完毕”，包括执行到代码末端而正常结束的情况，也包括产生异常，或者通过 break、return 等中途跳出的情况。只要使用 ensure，就可以实现和带代码块的 open 调用同样的功能（图 8）。

```
def open_close(path, mode, &block)
 f = open(path, mode)
 begin
   block.call(f)
 ensure
   f.close
 end
end
```

图 8　ensure 必定会被执行

ensure 的起源是来自 Lisp 的 unwind-protect 函数。这个函数名的意思是，当访问磁带设备出错时，防止（protect）出现磁带没有回卷（unwind）的情况。

其他语言中的异常处理

刚才我们讲了 Ruby 中的异常处理，当然，其他语言中也具备异常处理的功能。例如在 Java 中，对应关系是这样的：

```
begin → try
rescue → catch
ensure → finally
```

在 C++ 中 try 和 catch 是和上面相同的，不过没有 finally。在 C++ 中，可以通过栈对象的析构函数（函数结束时必定会被调用）来实现相当于 ensure 的功能。

Java 的检查型异常

Java 的异常处理具有其他语言所不具备的特性，即每个方法都需要显式地声明自己可能会产生什么样类型的异常。图 9 是 Java 中的方法定义（节选）。在数据类型、方法名和参数之后，有一段形如 throws 异常的代码，用于声明可能会产生的异常。

```
 void open_file() throws
FileNotFoundException {
   return new FileReader("/path/to/file");
 }
```

图 9　Java 的方法定义（带异常）

并且，在 Java 中调用某个方法时，对于在该方法定义中所声明的异常，如果没有用异常处理来进行捕获，且没有用 throws 继续抛给上层的话，就会产生一个编译错误，因为异常已经成为方法的数据类型的一部分了。像这样的异常被称为检查型异常（checked exception）。在广泛使用的编程语言中，Java 应该是第一个采用检查型异常的语言。

检查型异常可以由编译器对遗漏捕获的异常进行检查，从这个角度来说，这个功能相当有用，也是符合 Java 的一贯策略的，正如在 Java 中采用静态数据类型来主动规避类型不匹配的思路是一样的。

不过，检查型异常也遭到了一些批判。异常之所以被称为异常，本来就因为它很难事先预料到。明知如此，还非要在代码中强制性地事先对异常做好声明，以避免产生编译错误，这实在是太痛苦了。

在有些情况下，Java 的方法会抛出如 SQLException 和 IOException 这样的异常，尽管实际上这些错误跟数据库和文件没什么关系。很显然，这是由于在实现这些功能时所调用的方法抛出了这些异常，但将这些实现的详细信息展现给用户是完全没有必要的。

尽管如此，如果每次都一定要按照方法的含义去更换异常的类型，或者为了避免编译器出错而硬着头皮写代码去捕获异常，这就显得本末倒置了。数据类型的问题也是一样，碰到编译错误，也就是把编译器给“惹毛了”。如果说因为真正的程序错误惹毛了编译器也就算了，要是仅仅因为异常的类型稍稍不合就大发雷霆的话，那这个编译器也太神经过敏了。而且，如果只是为了迁就编译器就非要编写一大堆异常处理代码的话，那异常本身的便利性就全都白费了。

话说，大家千万别误会，检查型异常也是有优点的。只不过，从我个人来看，比起一个十分严格的，像对错误零容忍的老师一样的编译器来说，我还是更喜欢 Ruby 这样相对比较宽容的语言吧。

Icon 的异常和真假值

异常也有比较特别的用法，为此我们来介绍一种叫做 Icon 的语言。Icon 是由美国亚利桑那大学开发的,用于字符串模板匹配等处理的编程语言。它诞生于 1977 年,是一种非常古老的语言。

在 Icon 中，异常（在 Icon 中称为失败）是通过“假”来表示的。也就是说，当对表达式求值时，如果没有产生异常，则结果为真，反之则结果为假。因此，像：

```
if 表达式
```

这样的条件判断，并不是 Ruby 等一般语言中“表达式结果为真时”的判断方式，而是“表达式求值成功时（没有产生异常）”的意思。也就是说，像：

```
a < b
```

这样一个简单的表达式，在一般语言中它的判断方式为：将 a 和 b 进行比较，当 b 较大时为真，两者相等或 b 较小时为假。而在 Icon 中它的判断方式为：将 a 和 b 进行比较，两者相等或 b 较小时产生异常，否则返回 b 的值。因此，在 Icon 中：

```
a < b < c
```

这样的表达式是比较正当的。对这个表达式进行求值时，由于 $a < b$ 的比较结果为真时，表达式的求值结果为 b，则接下来会对 $b < c$ 进行求值。如果最开始的比较结果为假，则整个表达式的求值就失败了，后面的比较操作实际上并没有被执行。这种方式真的非常独特。

在 Ruby 等以真假来求值的语言中，要得到相同的结果，必须要写成这样：

```
a < b && b < c
```

说句题外话，在 Python 中其实也是可以写成：

```
a < b < c
```

这样的，不过这并不是说 Python 具备像 Icon 这样的运行模块，而只是其语法分析器可以识别连续的比较运算符，最终还是要将表达式转换成：

```
a < b && b < c
```

这样的形式。

在以异常为基础的 Icon 中，从文件中逐行读取内容并输出的程序写成下面这样：

```
while write(read())
```

好像语序有点奇怪吗？ Icon 中就是这样写的。

首先，read 函数从标准输入读取 1 行数据，当读取成功时则返回读取到的字符串。write 函数将通过参数得到的字符串写到标准输出。通过这样的方式，就完成了“读取一行内容并输出”的操作。

读懂这段程序的关键，在于将这个读取一行的操作作为 while 循环的条件判断来使用。Icon 的 while 语句的逻辑是“执行循环直到条件判断表达式失败”，因此，write(read()) 这个操作将被循环执行，直到失败为止，而在读取到文件末尾时，read 函数会失败，这个失败会被 while 语句的条件判断捕获，从而结束循环。习惯了一般语言的人，可能会感觉很异样，因为这个 while 循环并没有循环体，却可以执行所需的操作，不过当你明白了其中的逻辑，也就觉得顺理成章了。

此外，在 Icon 中，还有一种叫做 every 的控制结构，它可以对所有的组合进行尝试，直到失败为止。Icon 的这种求值方式，由于包含了“继续求值到达到某种目标为止”的含义，因此被称为目标导向求值（Goal-directed evaluation）。例如：

```
every write((1 to 3) + (2 to 3))
```

表示将 1 到 3 的数，和 2 到 3 的数，用不同的排列组合来输出它们的合，即 1+2、1+3、2+2、2+3、3+2、3+3，运行结果为：

```
3
4
4
5
5
6
```

在一般语言中，这样的运算需要通过两层循环来完成。运用异常和目标导向求值，可以在无显式循环的情况下，对排列组合运算进行描述，这一点实在是很有意思。

综上所述，在 Icon 中，异常和真假值的组合非常强大，应用范围也很广，颇具魅力。在最初设计 Ruby 的时候，我也曾经认真思考过，到底要不要采用 Icon 这样的真假求值机制，结果却还是采用了用 nil 和 false 表示“假”，其余都表示“真”这样的正统方式。当时，如果做出另一种不同的判断的话，也许 Ruby 这个语言的性质就会发生很大的改变呢。

Eiffel 的 Design by Contract

从异常这个角度来看，还有一种很有意思的语言，叫做 Eiffel[①]。Eiffel 中强调了一种称为 Design by Contract（契约式设计，简称 DbC）的概念。即所有的方法（在 Eiffel 中称为子程序）都必须规定执行前需要满足的条件和执行后需要满足的条件，当条件不能满足时，就会产生异常。

这样的思路，就是将对子程序的调用，看作是一种“只要兑现满足先验条件的约定，后验条件就必定得到满足”的契约。

Eiffel 的子程序定义代码如图 10 所示。Eiffel 中异常没有类型的区别，这也是强调 DbC 设计方针的结果，和其他的语言有所不同。

大家应该可以看出，Eiffel 的异常处理中所使用的保留字（ensure、rescue、retry），在 Ruby 中得到了继承。具体的含义可能有所不同，但 Ruby 开发早期确实参考了 Eiffel 中的保留字。

```
-- Eiffel中“--”开头的是注释
-- 方法定义
command is
 require
   -- 先验条件
 local
   -- 局部变量声明
 do
   -- 子程序正文
 ensure
   -- 后验条件
 rescue
   -- 异常处理
   -- 通过retry返回do重新执行
 end
```

图 10　Eiffel 的方法定义

异常与错误值

像 C 语言这样完全不支持错误处理的语言中，异常状况只能通过错误值来表示。那么，在具备异常功能的语言中，是不是所有的错误都可以通过异常来表示呢？

以我的一己之见，大部分情况下都应该使用异常，不过也有一些情况下用错误值更好。例如，在 Ruby 中也有一些情况是需要用错误值的。

对 Hash 的访问算是一个例子。在 Ruby 中，访问 Hash 时如果 key 不存在的话，并不会产生异常，而是会返回 nil（在 Python 中则会产生异常）。

① Eiffel 是一种面向对象编程语言，诞生于 1986 年，设计者是 Bertrand Meyer（1950—　）。

```
hash[key] # => 不存在时返回nil
```

也就是说，这要看对访问 Hash 时 key 不存在这一情况到底能做出何种程度的预计。如果 key 不存在的情况完全是超出预计的，错误就应该作为异常来处理；反之，如果 key 不存在的情况在某种程度上是预计范围内的，那么就应该返回错误值。

不过，在某些情况下，我们希望将 key 不存在的情况作为错误来产生异常，并且保证要将其捕获。在这种情况下，可以使用 Hash 类中的 fetch 方法，用这个方法的话，当 key 不存在时就会产生异常。

小结

对于程序员来说，错误处理虽然不希望发生，但也不能忽视，是个很麻烦的事情。异常处理功能就是为了将程序员进行错误处理的负担尽量减轻而产生的一种机制。21 世纪的编程语言中，绝大部分都具备了异常处理功能，我想这也是编程语言实现了进化的一个证据吧。

2.6 闭包

有一次，我参加了一个叫做“Ruby 集训”的活动，那是一个由想学习 Ruby 的年轻人参加的，历时 5 天 4 夜的 Ruby 编程学习活动，对参加者来说是一次非常宝贵的经验。第 1 天是入门培训，第 2 天将 Ruby 系统学习一遍，然后第 3 天到第 4 天分组各自制作一个相当规模游戏，并在最后一天进行展示，可以说是一次十分军事化的集训活动。我只到现场去了大概两天，不过那些勇于向高难度课题发起挑战的年轻人还是给我留下了深刻的印象。

在那次集训活动中，有一位参加者问：“闭包是什么？”担任讲师的是我的学生，不过他也没有做出准确的理解，因此借这个机会，我想仔细给大家讲一讲关于闭包的话题。

函数对象

有一些编程语言中提供了函数对象这一概念，我知道有些人把这个叫做闭包（Closure），但其实这种理解是不准确的，因为函数对象不一定是闭包。不过话说回来，要理解闭包，首先要理解函数对象，那么我们先从函数对象开始讲起吧。

所谓函数对象，顾名思义，就是作为对象来使用的函数。不过，这里的对象不一定是面向对象中所指的那个对象，而更像是编程语言所操作的数据这个意思。

例如，C 语言中，我们可以获取一个函数的指针，并通过指针间接调用该函数。这就是 C 语言概念中的对象（图 1）。

```
 1 #include <stdio.h>
 2 int two(int x) {return x*2;}
 3 int three(int x) {return x*3;}
 4
 5 int main(int argc, char **argv)
 6 {
 7   int (*times)(int);
 8   int n = 2;
 9
10   if (argc == 1) times = two;
11   else times = three;
12   printf("times(%d) = %d\n", n, times(n));
13 }
```

图 1　C 语言的函数对象

一般的 C 语言程序员应该不大会用到函数指针，因此我们还是讲解一下吧。

第 7 行，main 函数的开头有个不太常见的写法：

```
int (*times)(int);
```

这是对指针型变量 times 的声明，它的意思是：变量 times，是指向一个拥有一个 int 型参数，并返回 int 值的函数的指针。

第 10 行开始的 if 语句，意思是当传递给程序的命令行参数为零个时。当参数为零个时，将函数 two（的指针）赋值给变量 times；当存在一个以上的参数时，则将函数 three 的指针赋值给 times。

综上所述，当程序没有命令行参数时，则输出：

```
times(2) = 4
```

有命令行参数时，则输出：

```
times(2) = 6
```

到这里，大家应该对 C 语言中的函数指针有所了解了吧？

高阶函数

重要的是，这种函数对象对我们的编程有什么用。如果什么用都没有的话，那就只能是语言设计上的一种玩具罢了。

函数对象，也就是将函数作为值来利用的方法，其最大的用途就是高阶函数。所谓高阶函数，就是用函数作为参数的函数。光这样说大家可能不太明白，我们来通过例子看一看。

我们来设想一个对数组进行排序的函数。这个函数是用 C 语言编写的，在 API 设计上，应该写成下面这样：

```
void sort(int *a, size_t size);
```

函数接受一个大小为 size 的整数数组 a，并对其内容进行排序。

不过，这个 sort 函数有两个缺点。第一，它只能对整数数组进行排序；第二，排序条件无法从外部进行指定。例如，我们希望对整数数组进行逆序排序，或者是希望对一个字符串数组按 abc 顺序、辞典顺序进行排序等等，用这个函数就无法做到。也就是说，这个 sort 函数是缺乏通用性的。

用函数参数提高通用性

另一方面，在 C 语言标准库中，却提供了一个具有通用性的排序函数，它的名字叫 qsort，API 定义如图 2 所示。

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

图 2　qsort 函数

那么，这个通用排序函数 qsort 是如何克服上述两个缺点的呢？秘密就隐藏在 qsort 函数的参数中。

首先，我们来看看第 1 个参数 base，它的类型是 void*。sort 的第 1 个参数是限定为整数数组的，相比之下，qsort 的参数则表示可以接受任何类型的数组。这样就避免了对数组类型的限制。

接下来，第 2、第 3 个参数表示数组的大小。在 sort 中只传递了数组的大小（元素的数量），而 qsort 中的第 2 个参数 nmemb 表示元素数量，第 3 个参数 size 则表示每个元素的大小。这样一来，相对于只能对整数数组进行排序的 sort 函数来说，qsort 则可以对任何数据类型的数组进行排序。

不过，还有一个重要的问题，那就是如何对任意类型数组中的元素进行比较呢？要解决这个问题，就要靠 qsort 函数的第 4 个参数 compar 了。

compar 是指向一个带两个参数的函数的指针。这个函数接受数组中两个元素的指针，并以整数的形式返回比较结果。当两个元素相等时，返回 0，当 a 比 b 大时返回正整数，当 a 比 b 小时返回负整数。

qsort 的实际应用例如图 3 所示。在这里我们定义了一个名为 icmp 的函数，它可以对整数进行逆序比较。结果，qsort 函数就会将数组中的元素按降序（从大到小）排序。

```
#include <stdio.h>
#include <stdlib.h>

int icmp(const void *a, const void *b)
{
 int x = *(int*)a;
 int y = *(int*)b;

 if (x == y) return 0;
 if (x > y) return -1;
 return 1;
}

int main(int argc, char **argv)
{
 int ary[] = {4,7,1,2};
 const size_t alen = sizeof(ary)/sizeof(int);
 size_t i;

 for (i=0; i<alen; i++) {
   printf("ary[%d] = %d\n", i, ary[i]);
 }
 qsort(ary, alen, sizeof(int), icmp);
 for (i=0; i<alen; i++) {
   printf("ary[%d] = %d\n", i, ary[i]);
 }
}
```

图 3　qsort 函数的应用实例

大家现在应该已经明白了，qsort 函数是通过将另一个函数作为参数使用，来实现通用排序功能的。高阶函数这样的方式，通过将一部分处理以函数对象的形式转移到外部，从而实现了算法的通用化。

函数指针的局限

好，关于（C 语言的）函数指针以及将其用作参数的高阶函数的强大之处，我们已经讲过了，下面我们来讲讲它的局限吧。

作为例题，我们来设想一下，对结构体构成的链表（Linked list）及对遍历处理，用高阶函数来进行抽象化。

图 4 是用一般的循环和高阶函数两种方式对链表进行遍历的程序。图 4 的程序由于 C 语言性质的缘故显得很长，其本质的部分是从 main 函数第 38 行开始的。

从第 39 行开始的 while 语句没有使用高阶函数，而是直接用循环来实现的。受过良好训练的 C 语言程序员可能觉得没什么，不过要看懂 41 行的

```
l = l->next
```

等写法，需要具备关于链表内部原理的知识，其实这些涉及底层的部分，最好能够隐藏起来。

另一方面，第 43 行开始用到 foreach 函数的部分，则是非常清晰简洁的。只不过，受到 C 语言语法的制约，这个函数必须在远离循环体的地方单独进行定义，这是 C 语言函数指针的第一个缺点。大多数语言中，函数都可以在需要调用的地方当场定义，因此这个缺点是 C 语言所固有的。

不过和另一个重要的缺点相比，这第一个缺点简直算不上是缺点。如果运行这个程序的话，结果会是下面这样的。

```
node(0) = 3
node(1) = 2
node(2) = 1
node(3) = 0
node(?) = 3
node(?) = 2
node(?) = 1
node(?) = 0
```

前面 4 行是 while 语句的输出结果，后面 4 行是 foreach 的输出结果。while 语句的输出结果中，

可以显示出索引，而 foreach 的部分则只能显示“?”。这是因为和 while 语句不同，foreach 的循环实际上是在另一函数中执行的，因此无法从函数中访问位于外部的局部变量 i。当然，如果变量 i 是一个全局变量就不存在这个问题了，不过为了这个目的而使用副作用很大的全局变量也并不是一个好主意。因此，“对外部（局部）变量的访问”是 C 语言函数指针的最大弱点。

```
#include <stdio.h>
#include <stdlib.h>

struct node {                     /* 结构体定义 */
  struct node *next;
  int val;
};

typedef void (*func_t)(int);    /* 函数指针类型 */

void                              /* 循环用函数 */
foreach(struct node *list, func_t func)
{
  while (list) {
    func(list->val);
    list = list->next;
  }
}

void                              /* 循环主体函数 */
f(int n)
{
  printf("node(?) = %d\n", n);
}

main()                            /* main函数 */
{
  struct node *list = 0, *l;
  int i;
                                  /* 准备开始 */
  for (i=0; i<4; i++) {           /* 创建链表 */
    l = malloc(sizeof(struct node));
    l->val = i;
    l->next = list;
    list = l;
  }

  i = 0; l = list;                /* 例题主体 */
  while (l) {                     /* while循环 */
    printf("node(%d) = %d\n", i++, l->val);
    l = l->next;
  }
  foreach(list, f);               /* foreach循环 */
}
```

图 4　高阶函数循环

作用域：变量可见范围

现在我们已经了解了 C 语言提供的函数指针的缺点，于是，为了克服这些缺点而出现的功能，就是本次的主题——闭包。

我想现在大家已经理解了函数对象，下面我们来讲解一下闭包。话说，要讲解闭包，必须使用一种支持闭包的语言才行，因此在这里我们用 JavaScript 来讲解。肯定有人会问，为什么不用 Ruby 呢？关于这一点，我们稍后再讲。

首先，为了帮助大家理解闭包，我们先来介绍两个术语：作用域（Scope）和生存周期（Extent）。

作用域指的是变量的有效范围，也就是某个变量可以被访问的范围。在 JavaScript 中，保留字 var 所表示的变量声明所在的最内侧代码块就是作用域的单位（图 5），而没有进行显式声明的变量就是全局变量。作用域是嵌套的，因此位于内侧的代码块可以访问以其自身为作用域的变量，以及以外侧代码块为作用域的变量。

另外，大家别忘了创建匿名函数对象的语法。在 JavaScript 中是通过下面的语法来创建函数对象的：

```
function () {...}
```

图 5 中我们将匿名函数赋值给了一个变量，如果不赋值而直接作为参数传递也是可以的。当然，这个函数对象也有自己的作用域。

```
var a = 1;          // a是全局变量
function foo() {
 var b = 2;         // b只在foo中可见
 g = 3;             // g是全局变量
 var f = function () {
   var c = 4;       // c只在该函数内可见
   return a+b+c     // a、b和c都可见
 };
 f();               // 调用函数对象f
}
```

a、g 的作用域

b、f 的作用域

c 的作用域

图 5　JavaScript 中的作用域

由于 JavaScript 中可以直接定义函数对象，因此像图 4 那样应用 foreach 的程序，用 JavaScript 就可以更加直接地编写出来。将图 4 的本质部分用 JavaScript 来改写的程序如图 6 所示。

```
function foreach(list, func) {  // 循环高阶函数
 while (list) {
   func(list.val);
   list = list.next;
 }
}

var list = null;                // 变量声明
for (var i=0; i<4; i++) {       // list初始化
   list = {val: i, next: list};
}

var i = 0;                      // i初始化
// 从函数对象中访问外部变量
foreach(list, function(n){console.log("node("+i+") = "+n);i++;});
```

图 6　高阶函数循环

这里值得注意的是，作为 foreach 参数的函数对象，是可以访问在外部声明的变量 i 的。结果，C 语言版的 foreach 函数无法实现的索引显示功能，在这里就可以实现了。因此，从函数对象中能够对外部变量进行访问（引用、更新），是闭包的构成要件之一。

按照作用域的思路，可能大家觉得上述闭包的性质也是理所当然的。不过，如果我们加上另外一个概念——生存周期，结果可能就会出乎意料了。

生存周期：变量的存在范围

所谓生存周期，就是变量的寿命。相对于表示程序中变量可见范围的作用域来说，生存周期这个概念指的是一个变量可以在多长的周期范围内存在并被能够被访问。要搞清楚这个概念，我们还是得看看实例。

图 7 的例子是一个返回函数对象的函数，即 extent 这个函数的返回值是一个函数对象。函数对象会对 extent 中的一个局部变量 n 进行累加，并显示它的值。

```
function extent() {
 var n = 0;                     // 局部变量
 return function() {
    n++;                        // 对n的访问
    console.log("n="+n);
 }
}
f = extent();                    // 返回函数对象
f();                             //  n=1
f();                             //  n=2
```

图 7　变量的生存周期

那么，这个程序实际运行的情况会如何呢？

extent() 执行后会返回函数对象，我们将其赋值给一个变量。这个函数变量在每次被执行时，局部变量就会被更新，从而输出逐次累加的结果。

咦？这里不觉得有点怪吗？

局部变量 n 是在 extent 函数中声明的，而 extent 函数已经执行完毕了啊。变量脱离了作用域之后不是应该就消失了吗？不过，就这个运行结果来看，即便在函数执行完毕之后，局部变量 n 貌似还在某个地方继续存活着。

这就是生命周期。也就是说，这个从属于外部作用域中的局部变量，被函数对象给“封闭”在里面了。闭包（Closure）这个词原本就是封闭的意思。被封闭起来的变量的寿命，与封闭它的函数对象寿命相等。也就是说，当封闭这个变量的函数对象不再被访问，被垃圾回收器回收掉时，这个变量的寿命也就同时终结了。

现在大家明白闭包的定义了吧。在函数对象中，将局部变量这一环境封闭起来的结构被称为闭包。因此，C 语言的函数指针并不是闭包，JavaScript 的函数对象才是闭包。

闭包与面向对象

在图 7 的程序中，当函数每次被执行时，作为隐藏上下文的局部变量 n 就会被引用和更新。也就是说，这意味着函数（过程）与数据结合起来了。

“过程与数据的结合”是形容面向对象中的“对象”时经常使用的表达。对象是在数据中以方法的形式内含了过程，而闭包则是在过程中以环境的形式内含了数据。即，对象和闭包是同一事物的正反两面。所谓同一事物的正反两面，就是说使用其中的一种方式，就可以实现另一种方式能够实现的功能[①]。例如图 7 的程序，如果用 JavaScript 的面向对象功能来实现的话，就成了图 8 中的样子。

```
function extent() {
 return {val: 0,
         call: function() {
             this.val++;
             console.log("val="+this.val);
         }};
}
f = extent();                         // 返回对象
f.call();                             // val=1
f.call();                             // val=2
```

图 8　通过面向对象来实现

Ruby 的函数对象

到此为止，我们在例子中使用的语言都是 JavaScript，那为什么不用我最擅长的 Ruby 语言呢？下面我来说说理由吧。

① 准确地说，对象可以拥有多个过程，而闭包只能拥有一个。但是，闭包中可以将相当于过程名的符号作为参数进行传递，通过内部分支，实际上也可以提供过程的功能。（原书注）

最大的一个理由是，Ruby 语言中是没有函数这个概念的。作为纯粹面向对象的语言，Ruby 中的一切过程都是从属于对象的方法，而并不存在独立于对象之外的函数。但是，Ruby 有具备和函数对象相同功能的 Proc（过程）对象，在实际应用上和函数对象的用法是差不多的。不过，这样一来讲解就会变得很麻烦，因此我们便采用了具备简单函数对象功能的 JavaScript。

为了向大家演示一下 Ruby 也能实现和 JavaScript 相同的功能，我们将图 7 的程序用 Ruby 改写了一下，如图 9 所示。

```
def extent
 n = 0                          # 局部变量
 lambda {                       # 过程对象表达式
   n+=1                         # 对n的访问
   printf "n=%d\n", n
 }
end
f = extent();                    # 返回函数对象
f.call();                        # n = 1
f.call();                        # n = 2
```

图 9　Ruby 的变量生存周期

将图 7 和图 9 对比一下，值得注意的是，在 Ruby 中创建过程对象需要使用 lambda{…} 表达式，且调用过程对象不能只加上一对括号，而是必须通过 call 方法进行显式调用。

在 Ruby 1.9 中，为了对函数型编程提供支持，lambda 可以用 -> 表达式来替代，此外 call 方法的调用也可以省略成 f.() 的形式，只不过 f 后面的那个圆点还必须要写，这一点挺遗憾的。

Ruby 与 JavaScript 的区别

从函数这个角度来看，Ruby 和 JavaScript 的区别还是很大的，关于这一点我们来详细说说吧。

正如之前所讲过的，Ruby 中只有方法而没有函数，而过程对象是可以用类似函数的方式来使用的。由于过程对象并不是函数，因此需要调用 call 方法，但除此之外，像闭包等其他语言的函数对象所具备的性质，过程对象也都具备。另一方面，JavaScript 中有函数，自然可以作为对象来引用（图 7）。但是，JavaScript 中方法与函数的区别很模糊，同样一个函数，在作为通常函数调用时，和作为对象的方法调用时，this 的值会发生变化（图 10）。

```
f = function() {
 console.log(this);
}
# 直接调用f
f();                    # this为global上下文
obj = {foo: f};         # 将f变为方法
# 将f作为方法来调用
obj.foo();              # this为obj
```

图 10　JavaScript 的 this

Lisp-1 与 Lisp-2

Ruby 和 JavaScript 的区别还有一点，那就是访问方法成员的行为方式。例如，假设 Ruby 和 JavaScript 的程序中都有一个名为 obj 的对象，两者都拥有一个名为 m 的方法。这时，同样是访问：

```
obj.m
```

Ruby 和 JavaScript 的行为是有很大差异的。在 Ruby 中，这行代码表示对 m 方法进行无参数调用，而在 JavaScript 中则表示返回实现 m 方法的函数对象，而如果要进行无参数调用的话，括号是不能省略的，如：

```
obj.m()
```

也就是说，JavaScript 中由圆点所引导的访问代表对属性的引用，将函数作为属性值返回的就是方法，而加上括号就可以对其进行调用。

另一方面，Ruby 中圆点所引导的访问只不过是对方法的调用而已，加不加括号，是不影响方法调用这一行为的。在 Ruby 中，如果要获取实现该方法的过程对象，则需要使用 method 方法（表 1）。

表1　Ruby和JavaScript的方法访问

	Ruby	JavaScript
方法调用（无参数）	obj.m	obj.m()
方法调用（有参数）	obj.m(1)	obj.m(1)
方法获取	obj.method(:m)	obj.m

光从这张表来看，会给人一种 JavaScript 整体上比较简洁的印象，而实际上，JavaScript 对获取方法实现这一不会频繁执行的操作，反而赋予了一种较简短的记法，却无法像 Ruby 一样省略方法调用时的括号，因此很难说 JavaScript 的这种模式就一定比较好（当然，这里面也有本作者的私心）。

从整体来看，作为纯粹面向对象的语言，Ruby 将对方法的调用放在中心位置；相对而言，JavaScript 的面向对象功能，是由函数对象这一概念发展而来的。

Python 也采用了和 JavaScript 相同的手法。如果对一种原本并非为面向对象设计的语言添加面向对象功能的话，这是一种十分有效的手法。

类似这样的设计思想的差异，在 Lisp 中早就存在，这两种做法分别叫做 Lisp-1（JavaScript 风格）和 Lisp-2（Ruby 风格）。

在 Lisp 的方言中，Scheme 等是属于 Lisp-1 的，函数和变量的命名空间是相同的。Lisp-1 这个名称，貌似就是从命名空间唯一这一概念而来的。在 Scheme 中，函数就是一个存放对函数对象的引用的变量而已。

因此，可以像这样：

```
(display "hello world")
(define d display)
(d "hello world")
```

仅通过赋值操作就可以为函数定义别名。

此外，Lisp 的另一个方言 EmacsLisp 中，变量和函数分别拥有各自的命名空间。如果执行这样的赋值操作：

```
(echo "hello world")
(setq e echo)
```

就会产生一个错误：

```
undefined variable echo
```

这是由于虽然存在名为 echo 的函数，却不存在名为 echo 的变量。如果要在 EmacsLisp 上实现和上述 Scheme 的例子相同的操作，就需要这样写：

```
(fset 'e (symbol-function 'ech
o))
(e "foo")
```

symbol-function 用来通过名称获取函数实体，而 fset 将名称与函数实体进行关联。虽然 Scheme 的风格看上去比较简单，但获取函数实体这种操作，一般人是不会去做的，因此没有必要将这种操作定义得这么简单。当然，用 Lisp 的本来就不是一般人了吧，这一点我们就当没看见吧。

现在大家应该明白了，通过闭包，可以实现更加高度的抽象化。刚才我们介绍了 C、JavaScript、Ruby、Lisp 等各种语言中函数对象的实现手法，希望大家能够通过上面的介绍，对这些语言的设计者在设计语言时的思路有一个大致的理解。

“编程语言的过去、现在和未来”后记

在正文中，我对未来的编程语言进行了预测，认为对云计算和多核的支持是编程语言未来发展的趋势，作为计算的进化方向，让多个计算机（核心）协同工作这一点我认为是毫无疑问的。本书中也对多核环境下的编程（第 6 章“多核时代的编程”），以及在服务器端对多台计算机的编程（第 4 章“云计算时代的编程”）等话题进行了阐述。

然而，谈到编程语言的进化方向，老实说我也是有点雾里看花的感觉。今后到底是出现一种对多核和云计算在设计上就进行积极支持的语言，然后这种语言逐步流行起来呢，还是在现存语言的基础上，以库的形式不断添加对上述环境的支持呢？虽然自诩为编程语言方面的专家，但对我来说这依然是一个很难预测的话题。

例如，Erlang 是一种对并行、分散编程提供积极支持的语言，是由瑞典爱立信公司于 20 世纪 80 年代后半期开始开发的。这种语言的风格，如：

- 受 Prolog 的影响
- 动态，函数型语言
- 单一赋值，无循环
- 基于 Actor 的消息传递
- 高容错性

与以往的语言都有很大差别，但却趁着近来的发展趋势迅速走红。此外，无需显式指定就能够在内部实现并行计算可能性的 Haskell 等语言也值得关注。

但是，尽管 Erlang 和 Haskell 获得了广泛的关注，和当前多核、云计算的发展速度相比，它们的走红也只是一时的。这其中的原因，可能是因为在现有语言上增加一些功能就足够了，不需要全新的语言，也可能是因为 Erlang 和 Haskell 所提供的与以往不同的范式和编程模型，一般的程序员还无法适应。总之，现在这个时点是很难做出判断的。

因此，这个领域在今后还是非常值得关注的。

第 3 章 编程语言的新潮流

3.1 语言的设计

接下来，我们从语言设计的角度，来比较一下 Java、JavaScript、Ruby 和 Go 这 4 种语言。这几种语言看起来彼此完全不同，但如果选择一个合适的标准，就可以将它们非常清楚地进行分类，如图 1 所示。

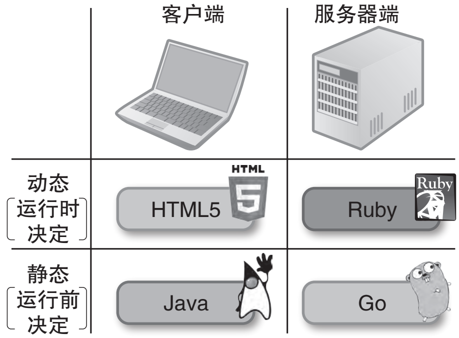


图 1　4 种语言的分类

Java 在最早的时候是作为客户端语言而诞生的。

JavaScript 是客户端语言的代表，Java 其实也在其黎明期作为客户端语言活跃过一段时间，应该有很多人还记得 Java Applet 这个名词。之后，Java 转型为服务器端语言的代表，地位也扶摇直上，但考虑到它的出身，这里还是将其分类为客户端语言。

另一个分类标准，就是静态和动态。所谓静态，就是不实际运行程序，仅通过程序代码的字面来确定结果的意思；而所谓动态，就是只有当运行时才确定结果的意思。静态、动态具体所指的内容有很多种，大体上来分的话就是运行模式和类型。这 4 种语言全都具备面向对象的性质，而面向对象本身就是一种包含动态概念的性质。不过，在这几种语言之中，Java 和 Go 是比较偏重静态一侧的语言，而 Ruby 和 JavaScript 则是比较偏重动态一侧的语言。

客户端与服务器端

首先，我们先将这些语言按照客户端和服务器端来进行分类。如前面所说，这种分类是以该语言刚刚出现时所使用的方式为基准的。

现在 Java 更多地被用作服务器端语言，而我们却将它分类到客户端语言中，很多人可能感到有点莫名其妙。Java 确实现在已经很少被用作客户端语言了，但是我们不能忘记，诞生于 1995 年的 Java，正是伴随嵌入在浏览器中的 Applet 技术而出现的。

Java 将虚拟机（VM）作为插件集成到浏览器中，将编译后的 Java 程序（Applet）在虚拟机上运行，这种技术当初是为了增强浏览器的功能。再往前追溯的话，Java 原本名叫 Oak，是作

为面向嵌入式设备的编程语言而诞生的。因此，从出身来看的话，Java 还是一种面向客户端的编程语言。

Java 所具备的 VM 和平台无关性字节码等特性，本来就是以在客户端运行 Applet 为目的的。在各种不同的环境下都能够产生相同的行为，这样的特性对于服务器端来说虽然也不能说是毫无价值，但是服务器环境是可以由服务提供者来自由支配的，因此至少可以说，这样的特性无法带来关键性的好处吧。另一方面，在客户端环境中，操作系统和浏览器都是千差万别，因此对平台无关性的要求一直很高。

Java 诞生于互联网的黎明时期，那个时候浏览器还不是电脑上必备的软件。当时主流的浏览器有 Mosaic 和 Netscape Navigator 等，[①]除此之外还有一些其他类似的软件，而 Internet Explorer 也是刚刚才崭露头角。

在那个充满梦想的时代，如果能开发出一种功能上有亮点的浏览器就有可能称霸业界。原 Sun Microsystems 公司[②]曾推出了一个用 Java 编写的浏览器 HotJava，向世界展示了 Applet 的可能性。然而，随着浏览器市场格局的逐步固定，他们转变了策略，改为向主流浏览器提供插件来集成 Java，从而对 Applet 的运行提供支持。

向服务器端华丽转身

然而，Java 自诞生之后，并未在客户端方面取得多大的成功，于是便开始着手进入服务器端领域。造成这种局面有很多原因，我认为其中最主要的原因应该是在 Applet 这个平台上迟迟没有出现一款杀手级应用（killer app）。

处于刚刚诞生之际的 Java 遭到了很多批判，如体积臃肿、运行缓慢等，不同浏览器上的 Java 插件之间也存在一些兼容性方面的问题，使得 Applet 应用并没有真正流行起来。在这个过程中，JavaScript 作为客户端编程语言则更加实用，并获得了越来越多的关注。当然，在那个时候 Java 已经完全确立了自己作为服务器端编程语言的地位，因此丧失客户端这块领地也不至于感到特别肉痛。

Java 从客户端向服务器端的转身可以说是相当成功的。与此同时，Sun Microsystems 和

① Mosaic 是世界上第一款真正流行的互联网浏览器软件，由美国国家超级计算机应用中心（National Center for Supercomputing Applications，NCSA）开发，1993 年发布，1997 年停止开发。Netscape Navigator 是由网景公司（Netscape）开发的一款互联网浏览器软件，1994 年发布，曾经一度是市场占有率最高的浏览器软件，后来它的地位被微软的 Internet Explorer 所取代。

② Sun Microsystems 于 2010 年以 74 亿美元被 Oracle 收购。

IBM 等公司着手对 JVM（Java VM）进行改良，使得其性能得到了改善，在某些情况下性能甚至超越了 C++。想想之前对 Java 性能恶评如潮的情形，现在 Java 能有这样的性能和人气简直就像做梦一样。

在服务器端获得成功的四大理由

由于我本人没有大规模实践过 Java 编程，因此对于 Java 在服务器端取得成功的来龙去脉，说真的并不是很了解。不过，如果让我想象一下的话，大概有下面几个主要的因素。

1. 可移植性

虽然服务器环境比客户端环境更加可控，但服务器环境中所使用的系统平台种类也相当多，如 Linux、Solaris、FreeBSD、Windows 等，根据需要，可能还会在系统上线之后更换系统平台。在这样的情况下，Java 所具备的"一次编写，到处运行"特性就显得魅力十足了。

2. 功能强大

Java 在服务器端崭露头角是在 20 世纪 90 年代末，那个时候的状况对 Java 比较有利。和 Java 在定位上比较相似的语言，即静态类型、编译型、面向对象的编程语言，属于主流的也就只有 C++ 而已了。

在 Java 诞生的 20 世纪 90 年代中期，正好是我作为 C++ 程序员开发 CAD 相关系统的时候。但当时 C++ 也还处于发展过程中，在实际的开发中，模板、异常等功能还无法真正得到运用。

相比之下，Java 从一开始就具备了垃圾回收（GC）机制，并在语言中内置了异常处理，其标准库也是完全运用了异常处理来设计的，这对程序员来说简直是天堂。毫无疑问，Java 语言的这些优秀特性，是帮助其确立服务器端编程语言地位的功臣之一。

3. 高性能

Java 为了实现其"一次编写，到处运行"的宣传口号，并不是将程序直接转换为系统平台所对应的机器语言，而是转换为虚拟 CPU 的机器语言"字节码"（Bytecode），并通过搭载虚拟 CPU 的模拟器 JVM 来运行。JVM 归根到底其实是在运行时用来解释字节码的解释器，理论上说运行速度应该无法与直接生成机器语言的原生编译器相媲美。

事实上，在 Java 诞生初期，确实没有达到编译型语言应有的运行速度，当时的用户经常抱怨 Java 太慢了，这样的恶评令人印象深刻。

然而，技术的革新是伟大的。随着各种技术的进步，现在 Java 的性能已经能够堪称顶级。

例如，有一种叫做 JIT（Just In Time）编译的技术，可以在运行时将字节码转换成机器语言，经过转换之后就可以获得和原生编译一样快的运行速度。在运行时进行编译，就意味着编译时间也会包含在运行时间里面。因此，优秀的 JIT 编译器会通过侦测运行信息，仅将需要频繁运行的瓶颈部分进行编译，从而大大削减编译所需的时间。而且，利用运行时编译，可以不用考虑连接的问题而积极运用内联扩展①，因此在某些情况下，运行速度甚至可以超过 C++。

在 Java 中，其性能提高的另一个障碍就是 GC。GC 需要对对象进行扫描，将不用的对象进行回收，这个过程和程序本身要进行的操作是无关的，换句话说，就是做无用功，因此而消耗的时间拖累了 Java 程序的性能。作为对策，在最新的 JVM 中，采用了并行回收、分代回收②等技术。

4. 丰富的库

随着 Java 的人气直升，应用逐渐广泛，Java 能够使用的库也越来越多。库的增加提高了开发效率，从而又反过来拉高了 Java 的人气，形成了一个良性循环。现在 Java 的人气已经无可撼动了。

客户端的 JavaScript

Applet 在客户端对扩展浏览器功能做出了尝试，然而它并不太成功。在浏览器画面中的一个矩形区域中运行应用程序的 Applet，并没有作为应用程序的发布手段而流行起来。

几乎是在同一时期出现的 JavaScript，也是一种集成在浏览器中的语言，但是它可以在一般的网页中嵌入程序逻辑，这一点是和 Java Applet 完全不同的方式，却最终获得了成功。

JavaScript 是由原 Netscape Communications 公司③开发的，通过 JavaScript，用户点击网页上的链接和按钮时，不光可以进行页面的跳转，还可以改写页面的内容。这样的功能十分便利，因此 Netscape Navigator 之外的很多浏览器都集成了 JavaScript。

随着浏览器的不断竞争和淘汰，当主流浏览器全部支持 JavaScript 时，情况便发生了变化。像 Google 地图这样的产品，整体的框架是由 HTML 组成的，但实际显示的部分却是通过

① 内联扩展（Inline expansion）是指让编译器直接将完整的函数体插入到每一个调用该函数的地方，从而提高函数调用的运行速度。

② 并行回收可以将 GC 放在单独的线程中运行，从而对程序本身的处理（基本上）不造成影响。分代回收是在扫描过程中忽略程序运行中一直存活的长寿对象，从而减少扫描工作量，降低 GC 开销的技术。（原书注）

③ 网景通信（Netscape Communications）于 1998 年被美国在线（AOL）收购，并于 2003 年解散。

JavaScript 来从服务器获取数据并显示出来，这样的手法从此开始流行起来。

在 JavaScript 中与服务器进行异步通信的 API 叫做 XMLHttpRequest，因此从它所衍生出的手法便被称为 Ajax（Asynchronous JavaScript and XML，异步 JavaScript 与 XML）。在美国有一种叫做 Ajax 的厨房清洁剂，说不定是从那个名字模仿而来的。

性能显著提升

目前，客户端编程语言中 JavaScript 已成为一个强有力的竞争者，伴随着 JavaScript 重要性的不断提高，对 JavaScript 引擎的投资也不断增加，使 JavaScript 的性能得到了显著改善。改善 JavaScript 性能的主要技术，除了和 Java 相同的 JIT 和 GC 之外，还有特殊化（Specialization）技术。

与 Java 不同，JavaScript 是一种动态语言，不带有变量和表达式的类型信息，针对类型进行优化是非常困难的，因此性能和静态语言相比有着先天的劣势，而特殊化就是提高动态语言性能的技术之一。

我们设想图 2 所示的这样一个 JavaScript 函数。这个函数是用于阶乘计算的，大多数情况下，其参数 n 应该都是整数。由于 JIT 需要统计运行时信息，因此 JavaScript 解释器也知道参数 n 大多数情况下是整数。

```
function fact(n) {
  if (n == 1) return 1;
  return n * fact(n-1);
}
```

图 2　JavaScript 函数

于是，当解释器对 fact 函数进行 JIT 编译时，会生成两个版本的函数：一个是 n 为任意对象的通用版本，另一个是假设 n 为整数的高速版本。当参数 n 为整数时（即大多数情况下），就会运行那个高速版本的函数，便实现了与静态语言几乎相同的运行性能。

除此之外，最新的 JavaScript 引擎中还进行了其他大量的优化①，说 JavaScript 是目前最快的动态语言应该并不为过。

JavaScript 在客户端称霸之后，又开始准备向服务器端进军了②。JavaScript 的存在感在将来应该会越来越强吧。

① 这些优化包括将 JavaScript 中本来以散列表（Hash table）形式实现的对象进行数组化，从而提高访问速度；通过内联化和特殊化相结合，实现和静态语言同等的对象访问速度等技术。（原书注）

② 在各种服务器端 JavaScript 的尝试中，最有力的一种就是 node.js。node.js 是将 Google Chrome 中搭载的高速 JavaScript 引擎 v8，与异步 I/O 相结合的产物。在第 6 章中我们将介绍 node.js。（原书注）

服务器端的 Ruby

客户端编程的最大问题，就是必须要求每一台客户端都安装相应的软件环境。在 Java 和 JavaScript 诞生的 20 世纪 90 年代后半，互联网用户还只局限于一部分先进的用户，然而现在互联网已经大大普及，用户的水平构成也跟着变得复杂起来，让每一台客户端都安装相应的软件环境，就会大大提高软件部署的门槛。

而相对的，在服务器端就没有这样的制约，可以选择最适合自己的编程语言。

在 Ruby 诞生的 1993 年，互联网还没有现在这样普及，因此 Ruby 也不是一开始就面向 Web 服务器端来设计的。然而，从 WWW 黎明期开始，为了实现动态页面而出现了通用网关接口（Common Gateway Interface，CGI）技术，而 Ruby 则逐渐在这种技术中得到了应用。

所谓 CGI，是通过 Web 服务器的标准输入输出与程序进行交互，从而生成动态 HTML 页面的接口。只要可以对标准输入输出进行操作，那么无论任何语言都可以编写 CGI 程序，这不得不归功于 WWW 设计的灵活性，使得动态页面可以很容易地编写出来，也正是因为如此，使得 WWW 逐渐风靡全世界。

在 WWW 中，来自 Web 服务器的请求信息是以文本的方式传递的，反过来，返回给 Web 服务器的响应信息也是以文本（HTML）方式传递的，因此擅长文本处理的编程语言就具有得天独厚的优势。于是，脚本语言的时代到来了。以往只是用于文本处理的脚本语言，其应用范围便一下子扩大了。

早期应用 CGI 的 Web 页面大多是用 Perl 来编写的，而作为“Better Perl”的 Ruby 也随之逐步得到越来越多的应用。

Ruby on Rails 带来的飞跃

2004 年，随着 Ruby on Rails 的出现，使得 Web 应用程序的开发效率大幅提升，也引发了广泛的关注。当时，已经出现了很多 Web 应用程序框架，而 Ruby on Rails 可以说是后发制人的。Ruby on Rails 的特性包括：

- 完全的 MVC 架构
- 不使用配置文件（尤其是 XML）
- 坚持简洁的表达
- 积极运用元编程
- 对 Ruby 核心的大胆扩展

基于这些特性，Ruby on Rails 实现了很高的开发效率和灵活性，得到了广泛的应用。可以说，Ruby 能拥有现在的人气，基本上都是 Ruby on Rails 所作出的贡献。

目前，作为服务器端编程语言，Ruby 的人气可谓无可撼动。有一种说法称，以硅谷为中心的 Web 系创业公司中，超过一半都采用了 Ruby。

但这也并不是说，只要是服务器端环境，Ruby 就一定可以所向披靡。在规模较大的企业中，向网站运营部门管理的服务器群安装软件也并不容易。实际上，在某个大企业中，曾经用 Ruby on Rails 开发了一个面向技术人员的 SNS，只用很短的时间就完成搭建了，但是等到要正式上线的时候，运营部门就会以“这种不知道哪个的家伙开发的，也没经过第三方安全认证的 Ruby 解释器之类的软件，不可以安装在我们数据中心的主机上面”这样的理由来拒绝安装，这真是相当头疼。

不过，开发部门的工程师们并没有气馁，而是用 Java 编写的 Ruby 解释器 JRuby，将开发好的 SNS 转换为 jar 文件，从而使其可以在原 Sun Microsystems 公司的应用程序服务器 GlassFish 上运行。当然，JVM 和 GlassFish 都已经在服务器上安装好了，这样一来运营方面也就没有理由拒绝了。多亏了 JRuby，结局皆大欢喜。

JRuby 还真是在关键时刻大显身手呢。

服务器端的 Go

Go 是一种新兴的编程语言，但它出身名门，是由著名 UNIX 开发者罗勃 · 派克和肯 · 汤普逊[①]开发的，因此受到了广泛的关注。

Go 的诞生背景源于 Google 公司中关于编程语言的一些问题。在 Google 公司中，作为优化编程环境的一环，在公司产品开发中所使用的编程语言，仅限于 C/C++、Java、Python 和 JavaScript。实际上也有人私底下在用 Ruby，不过正式产品中所使用的语言仅限上述 4 种。[②]

这 4 种语言在使用上遵循着一定的分工：客户端语言用 JavaScript，服务器端语言用脚本系的 Python，追求大规模或高性能时用 Java，文件系统等面向平台的系统编程用 C/C++。在这些语言中，Google 公司最不满意的就是 C/C++ 了。

---

① 罗勃 • 派克（Rob Pike，1956—　）是加拿大程序设计师，早年为贝尔实验室的 UNIX 小组成员，曾参与设计贝尔实验室 9 号计划（Plan 9）、Inferno 操作系统和 Limbo 编程语言，目前就职于 Google 公司。肯 • 汤普逊（Ken Thompson，1943—　）是美国计算机科学家，曾参与设计 Plan 9、B 语言、C 语言，并于 1983 年获得图灵奖。

② 此外还有一些内部专用的语言，例如，为了让用于处理 Web 爬虫所抓取的大量数据的 MapReduce 运行更高效，使用了专用语言 Sawzall。（原书注）

和其他一些编程语言相比，C/C++ 的历史比较久，因此不具备像垃圾回收等最近的语言所提供的编程辅助功能。因此，由于开发效率一直无法得到提高，便产生了设计一种“更好的”系统编程语言的需求。而能够胜任这一位置的，正是全新设计的编程语言 Go。

Go 具有很多特性，（从我的观点来看）比较重要的有下列几点：

- 垃圾回收
- 支持并行处理的 Goroutine[①]
- Structural Subtyping（结构子类型）

关于最后一点 *Structural Subtyping*，我们会在后面对类型系统的讲解中进行说明。

静态与动态

刚才我们已经将这 4 种语言，从客户端、服务器端的角度进行了分类。接下来我们再从动态、静态的角度来看一看这几种语言。

正如刚才所讲过的，所谓静态，就是无需实际运行，仅根据程序代码就能确定结果的意思；而所谓动态，则是只有到了运行时才能确定结果的意思。

不过，无论任何程序，或多或少都包含了动态的特性。如果一个程序完全是静态的话，那就意味着只需要对代码进行字面上的分析，就可以得到所有的结果，这样一来程序的运行就没有任何意义了。例如，编程计算 6 的阶乘，如果按照完全静态的方式来编写的话，应该是下面这样的：

```
puts "720"
```

不过，除非是个玩具一样的演示程序，否则不会开发出这样的程序来。在实际中，由于有了输入的数据，或者和用户之间的交互，程序才能在每次运行时都能得到不同的要素。

因此，作为程序的实现者，编程语言也多多少少都具备动态的性质。所谓动态还是静态，指的是这种语言对于动态的功能进行了多少限制，或者反过来说，对动态功能进行了多少积极的强化，我们所探讨的其实是语言的这种设计方针。

① Goroutine 是 Go 特有的一个术语，简单来说就是轻量版的线程。随着多核处理器的普及，并发编程的重要性不断提高，然而 C/C++ 在语言层面并不支持并发编程。在内核层面可以使用线程（pthread 等），但用起来并没有那么方便。在 Go 中，通过在语言层面所提供的支持，就很好地支持了系统编程层面中对并发编程的有效利用。（原书注）

例如，在这里所列举的 4 种编程语言都是面向对象的语言，而面向对象的语言都会具备被称为多态（Polymorphism）或者动态绑定的动态性质。即，根据存放在变量中的对象的实际性质，自动选择一种合适的处理方式（方法）。这样的功能可以说是面向对象编程的本质。

属于动态的编程语言，其动态的部分，主要是指运行模式和类型。这两者是相互独立的概念，但采用动态类型的语言，其运行模式也具有动态的倾向；反之也是一样，在静态语言中，运行模式在运行时的灵活性也会受到一定的限制。

动态运行模式

所谓动态运行模式，简单来说，就是运行中的程序能够识别自身，并对自身进行操作。对程序自身进行操作的编程，也被称为元编程[①]（Metaprogramming）。

在 Ruby 和 JavaScript 中，元编程是十分自然的，比如查询某个对象拥有哪些方法，或者在运行时对类和方法进行定义等等，这些都是理所当然的事。

另一方面，在 Java 中，类似元编程的手法，是通过“反射 API”来实现的。虽然对类进行取出、操作等功能都是可以做到的，但并非像 Ruby 和 JavaScript 那样让人感到自由自在，而是“虽然能做到，但一般也不会去用”这样的感觉吧。

Go 也是一样。在 Go 中，通过利用 reflect 包可以获取程序的运行时信息（主要是类型），但是（在我所理解的范围内）无法实现进一步的元编程功能。而之所以没有采用比 Java 更进一步的动态运行模式，恐怕是因为这（可能）在系统编程领域中必要性不大，或者是担心对运行速度产生负面影响之类的原因吧。

何谓类型

从一般性的层面来看，类型[②]指的是对某个数据所具有的性质所进行的描述。例如，它的结构是怎样的，它可以进行哪些操作，等等。动态类型的立场是数据拥有类型，且只有数据才拥有类型；而静态类型的立场是数据拥有类型，而存放数据的变量、表达式也拥有类型，且类型是在编译时就固定的。

① 程序“对自身进行操作”也被称为“反射”（Reflection）。在英语中，Reflection 一词有“对自己进行反省”的意思。同样的意思，在 Ruby 大多被称为元编程，而在 Java 中则大多被称为反射。（原书注）

② 其实，类型这个话题，在现在许多计算机科学论文所设计的领域中，算是比较热门的。不过在这里，我们不去探讨那些最尖端的话题。说实话，我的数学素养实在不行，对于那些通篇都是数学公式的类型理论方面的论文，实在是无法理解，因此很遗憾，我也就没办法讲解这些内容了。（原书注）

然而，即便是静态类型，由于面向对象语言中的多态特性，也必须具备动态的性质，因此需要再追加一条规则，即实际的数据（类型），是静态指定的类型的子类型。所谓子类型（Subtype），是指具有继承关系，或者拥有同一接口，即静态类型与数据的类型在系统上“拥有同一性质”。

静态类型的优点

动态类型比较简洁，且灵活性高，但静态类型也有它的优点。由于在编译时就已经确定了类型，因此比较容易发现 bug。当然，程序中的 bug 大多数都是与逻辑有关的，而单纯是类型错误而导致的 bug 只是少数派。不过，逻辑上的错误通常也伴随着编译时可以检测到的类型不匹配，也就是说，通过类型错误可以让其他的 bug 显露出来。

除此之外，程序中对类型的描述，可以帮助对程序的阅读和理解，或者可以成为关于程序行为的参考文档，这可以说是一个很大的优点。

此外，通过静态类型，可以在编译时获得更多可以利用的调优信息，编译器便可以生成更优质的代码，从而提高程序的性能。然而，通过 JIT 等技术，动态语言也可以获得与原生编译的语言相近的性能，这也说明，在今后静态语言和动态语言之间的性能差距会继续缩小。

动态类型的优点

相对而言，动态类型的优点，就在于其简洁性和灵活性了。

说得极端一点的话，类型信息其实和程序运行的本质是无关的。就拿阶乘计算的程序来说，无论是用显式声明类型的 Java 来编写（图 3），还是用非显式声明类型的 Ruby 来编写（图 4），其算法都是毫无区别的。然而，由于多了关于类型的描述，因此在 Java 版中，与算法本质无关的代码的分量也就增加了。

```
class Sample {
  private static int fact(int n) {
    if (n == 1) return 1;
    return n * fact(n - 1);
  }
  public static void main(String[] argv) {
    System.out.println("6!="+fact(6));
  }
}
```

图 3　Java 编写的阶乘程序

```
def fact(n)
  if n == 1
    1
  else
    n * fact(n - 1)
  end
end
print "6!=", fact(6), "\n"
---
```

图 4　Ruby 编写的阶乘程序

而且，类型也带来了更多的制约。图 3、图 4 中所示的程序对 6 的阶乘进行了计算，但如果这个数字继续增大，Java 版对超过 13 的数求阶乘的话，就无法正确运行了。图 3 的程序中，fact 方法所接受的参数类型显式声明为 int 型，而 Java 的 int 为 32 位，即可以表示到接近 20 亿的整数。如果阶乘的计算结果超出这个范围①，就会导致溢出。

当然，由于 Java 拥有丰富的库资源，用 BigInteger 类就可以实现无上限的大整数计算，但这就需要对上面的程序做较大幅度的改动。而由于计算机存在“int 的幅度为 32 位”这一限制，就使得阶乘计算的灵活性大大降低了。

另一方面，Ruby 版中则没有这样的制约，就算是计算 13 的阶乘，甚至是 200 的阶乘，都可以直接计算出来，而无需担心如 int 的大小、计算机的限制等问题。

其实这里还是有点小把戏的。同样是动态语言，用图 1 中的 JavaScript 来计算 200 的阶乘就会输出 Infinity（无穷大）。其实，JavaScript 的数值是浮点数，因此无法像 Ruby 那样支持大整数的计算。也就是说，要不受制约地进行计算，除了类型的性质之外，库的支持也是非常重要的。

有鸭子样的就是鸭子

在动态语言中，一种叫做鸭子类型（Duck Typing）的风格被广泛应用。鸭子类型这个称谓，据说是从下面这则英语童谣来的：

If it walks like a duck and quacks like a duck, it must be a duck.（如果像鸭子一样走路，像鸭子一样呱呱叫，则它一定是一只鸭子）

从这则童谣中，我们可以推导出一个规则，即如果某个对象的行为和鸭子一模一样，那无论它真正的实体是什么，我们都可以将它看做是一只鸭子。也就是说，不考虑某个对象到底是哪一个类的实例，而只关心其拥有怎样的行为（拥有哪些方法），这就是鸭子类型。因此，在程序中，必须排除由对象的类所产生的分支。

这是由“编程达人”大卫・托马斯（Dave Thomas）所提出的。

例如，假设存在 log_puts(out, mesg) 这样一个方法，用来将 mesg 这个字符串输出至 out 这个输出目标中。out 需要指定一个类似 Ruby 中的 IO 对象，或者是 Java 中的 OutPutStream 这样的对象。在这里，本来是向文件输出的日志，忽然想输出到内存的话，要怎么办呢？比如说我想将日志的输出结果合并成一个字符串，然后再将它取出。

① 13 的阶乘结果为 6227020800，而 Java 中 int 型的表示范围为 -2147483648 到 2147483647。

在 Java 等静态语言中，out 所指定的对象必须拥有共同的超类或者接口，而无法选择一个完全无关的对象作为输出目标。要实现这样的操作，要么一开始就事先准备这样一个接口，要么重写原来的类，要么准备一个可以切换输出目标的包装对象（wrapper object）。无论如何，如果没有事先预计到需要输出到内存的话，就需要对程序进行大幅的改动了。

如果是采用了鸭子类型风格的动态语言，就不容易产生这样的问题。只要准备一个和 IO 对象具有同样行为的对象，并将其指定为 out 的话，即便不对程序进行改动，log_puts 方法能够成功执行的可能性也相当大。实际上，在 Ruby 中，确实存在和 IO 类毫无继承关系，但和 IO 具有同样行为的 StringIO 类，用来将输出结果合并成字符串。

动态类型在编译时所执行的检查较少，这是一个缺点，但与此同时，程序会变得更加简洁，对于将来的扩展也具有更大的灵活性，这便是它的优点。

Structural Subtyping

在 4 种语言中最年轻的 Go，虽然是一种静态语言，但却吸取了鸭子类型的优点。Go 中没有所谓的继承关系，而某个类型可以具有和其他类型之间的可代换性，也就是说，某个类型的变量中是否可以赋予另一种类型的数据，是由两个类型是否拥有共同的方法所决定的。例如，对于“A 型”的变量，只要数据拥有 A 型所提供的所有方法，那么这个数据就可以赋值给该变量。像这样，以类型的结构来确定可代换性的类型关系，被称为结构子类型（Structural Subtyping）；另一方面，像 Java 这样根据声明拥有继承关系的类型具有可代换性的类型关系，被称为名义子类型（Nominal Subtyping）。

在结构子类型中，类型的声明是必要的，但由于并不需要根据事先的声明来确定类型之间的关系，因此就可以实现鸭子类型风格的编程。和完全动态类型的语言相比，虽然增加了对类型的描述，但却可以同时获得鸭子类型带来的灵活性，以及静态编译所带来了类型检查这两个优点，可以说是一个相当划算的交换。

小结

在这里，我们对 Ruby、JavaScript、Java、Go 这 4 种语言，从服务器端、客户端，以及静态、动态这两个角度来进行了对比。这 4 种语言由于其不同的设计方针，而产生出了不同的设计风格，大家是否对此有了些许了解呢？

不仅仅是语言，其实很多设计都是权衡的结果。新需求、新环境，以及新范式，都催生出新的设计。而学习现有语言的设计及其权衡的过程，也可以为未来的新语言打下基础。

3.2 Go

2009 年 11 月，Google 发布了一种名为 Go 的新语言，在世界范围内引发了轰动。下面我从一个编程语言设计者的角度，来展望一下 Go 这个新兴的编程语言。

作为一种新的编程语言，Go 宣扬了图 1 中的这些关键字。首先，我们先来看看这些关键字到底是什么意思吧。

```
（1）New（新的）
（2）Experimental（实验性的）
（3）Concurrent（并发的）
（4）Garbage-collected (带垃圾回收的)
（5）Systems（系统）
（6）Language
```

图 1　Go 的关键字

New（新的）

几乎每一个新的编程语言在发布的时候都会被问及这样一个问题：“为什么要创造一个新语言呢？”Ruby 发布的当时也有很多人这样问过，我给出的是“只是因为想做做看而已啊”这么个不着调的回答。不过 Go 的开发者们说，这个新语言是由于对现有语言的不满才诞生出来的。

在这 10 年中，有很多新的编程语言相继诞生，并获得一定程度的应用，其中大多数都是以 Ruby 为代表的动态语言，但是，能触及 C 和 C++ 领域的新的系统编程语言却迟迟没有出现。

另一方面，编程语言所面临的状况也在不断发生变化，如网络的普及、多核、大规模集群等，而重视性能的系统编程语言却没有对这样的变化做出应对。Go 语言的开发者们主张，正是因为这样的局面，才使得创造一种开发效率更高的系统编程语言变得十分必要。

Experimental（实验性的）

一种编程语言从出现到实用化所要经历的时间之长，超乎普通人的想象。以 Ruby 为例，从开始开发到发布用了 3 年左右的时间，而到了在程序员圈子中拥有一定知名度则又花了 4 年的时间，而再到通过 Ruby on Rails 而走红，则又花了 5 年的时间。

相比之下，从 2007 年末开始开发的 Go，只经过了 2 年左右的开发，在发布之时就获得了全世界的关注，我表示实在是羡慕之极。但即便如此，Go 所吸收的那些新概念是否能真正被世

界所接受，现在还是个未知数。从这个意义上来看，应该说它还只是一种实验性的语言。

Concurrent（并发的）

在21世纪的今天，并发编程变得愈发重要。需要同时处理大量并发访问的网络应用程序，本来就更加适合并发编程，而对于不断增大的处理信息量，分布式并发编程又是一个很好的解决方案，因而备受期待。

此外，要最大限度地利用多核，甚至是超多核（Many-core）环境的CPU性能，并发编程也显得尤为重要。

因此，为了实现更高开发效率的并发编程，编程语言本身也必须具备支持并发编程的功能，这已经成为一种主流的设计思路。近年来，像Erlang这样以并行计算为中心的编程语言受到了广泛的关注，也正是由于上述背景所引起的。

然而，当前主流的系统编程语言中，并没有哪种语言在语言规格层面上考虑到了并发编程。我想，正是这一点成为了Go开发的契机。

Garbage-collected（带垃圾回收的）

将不需要的对象自动进行回收，从而实现对内存空间的循环利用，这种垃圾回收（GC）机制在40多年前出现的Lisp等编程语言中已经是常识了。在需要大量操作对象的程序中，对于某个对象是否还要继续使用，是很难完全由人来把握和判断的。然而，如果对象的管理出现问题，便会导致十分严重的bug。

如果忘记对不需要的对象进行释放，程序所占用的内存容量就会不断增大，从而导致内存泄漏（Memory leak）bug；反过来，如果释放了仍然在使用中的对象，就会导致内存空间损坏的悬空指针（Dangling pointer）bug。

这两种bug都有一个特点，就是出问题的地方，和实际引发问题的地方往往距离很远，因此很难被发现和修复。所以我认为，在具备一定面向对象功能的编程语言中，GC是不可或缺的一个机制。

使GC走进普通的编程领域，并得到广泛的认知，不得不说是Java所带来的巨大影响。在Java之前，大家对GC的主流观点，要么认为它在性能上有问题，要么认为它在系统编程中是不需要的。像C++这样的系统编程语言中，没有提供GC机制，应该也是出于这个原因吧。

然而，现在情况变了。作为 21 世纪的系统编程语言，Go 具备了 GC 机制，从而减轻了对象管理的消耗，程序员的负荷也跟着减轻，从而使得开发效率得到了提高。

Systems（系统）

刚刚我们不断提到系统编程语言这个说法，那么系统编程语言到底指的是怎样一类编程语言呢?

至于严格的定义，其实我也不是十分清楚，不过从印象来说，应该指的是可以用来编写操作系统的，对性能十分重视的语言吧。从定位上来说，应该说是 C 和 C++ 所覆盖的那片领域。

的确，在这个领域最广泛使用的语言中，即便是最新的 C++（1983 年）也绝不能算是“新”了。而无法编译出可直接运行的代码（原本在设计上就不会编译出这样的代码）的 Java，又很难用作系统编程语言，而且 Java 发布于 1995 年，到现在也已经过了 10 多年了。

更进一步说，由于 Java 本身就是设计为在 JVM 中运行的，因此即便通过 JIT 等最新技术实现了高速化，我觉得也很难称其为是一种系统编程语言。

在 Google 中，由于对海量数据和大规模集群处理有较大的需求，因此便愈发需要一种高性能的编程语言。然而，为了避免使用过多种编程语言所造成的管理成本上升，Google 公司对官方项目中能够使用的语言进行了严格的限制，只有 C、C++、Java、JavaScript 和 Python 这 5 种[①]。

用于基础架构等系统编程的 C 和 C++、兼具高开发效率和高性能的 Java、用于客户端编程的 JavaScript，再加上高开发效率的动态语言 Python，我认为这是一组十分均衡的选择。

不过，仔细看看的话，用于系统编程的 C 和 C++ 则显得有些古老，对于最近获得广泛认知的，从语言层面对开发效率的支持机制（如 GC 等）显得不足。Go 的出现，则为这一领域带来了一股清新的风，也可以说，Go 是 Google 表达对于系统编程语言不满的一个结果。

Go 的创造者们

领导 Go 项目的，主要有下面这些人：罗勃·派克（Rob Pike）、肯·汤普逊（Ken Thompson）、

① 不过，在非官方的项目中就没有这样的限制，比如在 Google 公司内部也有很多 Ruby 程序员。貌似他们在官方项目中使用 Java 和 Python，而在非官方项目中则使用 Ruby。

Robert Griesemer、Ian Lance Taylor、Russ Cox。其中，罗勃·派克和肯·汤普逊是超级名人。肯·汤普逊是曾经最早创造 UNIX 的人，也是参与过 B 语言和 Plan 9[①]操作系统开发的传说中的黑客。

对我来说，罗勃·派克和布莱恩·柯林汉[②]合作的名著《UNIX 编程环境》给我留下了很深的印象，除此之外，罗勃·派克在 AT&T 贝尔实验室也贡献了诸多成果，如在 Plan 9 的开发中扮演了重要的角色。要说他离我们最近的一项功绩，莫过于 UTF-8 的开发了，这也是和肯·汤普逊的共同成果。如果没有他们的话，估计现在世界已经被那个超烂的 UTF-16 占领了吧，想到这里，我不禁充满了感激之情。

虽然可能有私情的成分，但这样的开发者所创造出的 Go，一定是颇受 UNIX，特别是 C 语言影响的，甚至可以说它就是现代版的 C 语言。因此，下面我们就通过和 C 语言的对比，为大家介绍一下 Go。

Hello World

Hello World 可以说是介绍编程语言的文章所必需的，如图 2 所示。这里面“世界”两个字并不是我故意给写成汉字的，而是罗勃·派克原始的 Hello World 程序就是这么写的。也许是为了证明罗勃·派克是开发者之一的缘故吧，这段程序表明，只要使用 UTF-8 字符串，就可以自由驾驭 Unicode。不过，貌似标识符还是只能用英文和数字。

```
package main ←—— 这段程序属于名为“main”的包
import "fmt" ←—— 使用名为“fmt”的包

func main () {
 fmt.Printf("Hello, 世界\n"); ←—— 使用fmt包中的Printf函数
}
```

图 2　用 Go 编写的 Hello World

由于只能引用公有函数，因此包名的圆点后面跟着的标识符总是大写字母开头。

在我的印象中，Go 和 C 语言果然还是很相似的。当然也有一些不同之处，例如 package 和 import 等对包系统的定义，以及末尾的分号可以省略等等。

Printf 中的 P 是大写字母，这一点挺引人注目的，其实它的背后代表了一条规则，即大写字母开头的名称表示可以从包外部访问的公有对象，而小写字母开头的名称表示只能从包内部访

① Plan 9，全称 Plan 9 from Bell Labs（贝尔实验室九号计划），是贝尔实验室从 20 世纪 80 年代中期到 2002 年为止，以研究 UNIX 后续可能性为主要目的而开发的操作系统。

② 布莱恩·柯林汉（Brian Kernighan，1942—　）是一位加拿大计算机科学家，曾任职于贝尔实验室，是 UNIX 的开发者之一。现在普林斯顿大学任教。

问的私有对象。由于有了这条规则，Go 中几乎所有的方法名都是大写字母开头的。

Go 的控制结构

下面我们来更加深入地了解一下 Go 吧。

首先我们从控制结构开始看。Go 主要的控制结构有 if、switch 和 for 三种，而并没有 while，while 用 for 代替了。我们先来讲一下 if 结构。

Go 的 if 结构语法规则如图 3 所示。

其中 else if 的部分可以有任意个。Go 的 if 结构和 C 的 if 结构很相似，不过有几点区别。

```
if 条件1 {程序体1}
  else if 条件2 {程序体2}
else {程序体3}
```

图 3　Go 的控制结构

首先，和 C 语言不同，Go 中 if 等结构中的条件部分并不用括号括起来，而相应地，程序体的部分则是必须用花括号括起来的。

还有一点不是很明显，那就是“必须括起来”这条规则实际上非常重要。C 语言的规则是这样的：当程序体包含多行代码时，需要用花括号括起来使其成为一体。但这样的规则下，if 结构的语法便会产生歧义。

例如：

```
if (条件) if (条件) 语句 else 语句
```

这样的 C 语言程序，到底是解释为：

```
if (条件) {
  if (条件) 语句
  else 语句
}
```

还是解释为：

```
if (条件) {
   if (条件) 语句
  }
else 语句
```

貌似很难抉择。

这个问题被称为“悬挂 else 问题”。在 C 语言中，虽然存在“有歧义的 else 属于距离较近的 if 语句”这样一条规则，但还是避免不了混乱的发生。

然而，如果有了“程序体必须用花括号括起来”这条规则，就不会产生这样的歧义了。从这个角度来看，不允许省略花括号着实是一个好主意。在其他广泛应用的主流语言中，Perl 的语法也不允许省略花括号。

说句题外话，Ruby 在控制结构的划分中，没有使用花括号，而是使用 end，理由也是一样的。像 Ruby 这样使用 end 的语言中，也可以避免因悬挂导致的歧义。

Go 的 if 语句还有一点与 C 语言不同，那就是在条件部分可以允许使用初始化语句，具体示例如图 4 所示。

```
if v = f(); v < 10 {
  fmt.Printf("%d < 10\n", v);
} else {
  fmt.Printf("%d >= 10", v);
}
```

图 4　条件部分可以使用初始化语句

将初始化语句移动到 if 结构的前面，意思也不会发生变化，但将初始化语句放在条件部分中，可以更加强调是对 f() 的返回值进行判断这一意图。我们后面要讲到的“逗号 OK”形式中，这种初始化方式也非常奏效。

switch 也是从 C 语言来的，但也有些微妙的差异。和 if 结构一样，条件表达式不用括号括起来，而程序体必须用花括号括起来（图 5）。

```
switch a {
 case 0: fmt.Printf("0");
 default: fmt.Printf("非0");
}
```

图 5　Go 的 switch 结构

当没有满足条件的 case 时则执行 default 部分，这一点和 C 语言是相同的。但是，和 C 语言相比，Go 的 switch 结构还存在下面这些差异：

- ❑ 即便没有 break，分支也会结束
- ❑ case 中可以使用任意的值
- ❑ 分支条件表达式可以省略

switch 中的 break 语法很诡异，堪称 C 语言中最大的谜，趁这个机会正好改一改。在上面的例子中，也不存在用于分隔的 break 呢。相应地，case 则可以并列多个条件。

虽说 Go 从 C 语言中继承了很多东西，但也没有必要连这些也一起继承过来。然而，Go 中却追加了一条新的语法，即 case 的程序体以 fallthrough 语句结束时，会进入下面一个分支。这样真的有必要吗？我觉得这只是一种对 C 语言单纯的怀念而已吧。

C 语言的 switch 中，case 可以接受的值只能是整数、字符、枚举等编译时已经确定的值，这应该是考虑了为用表实现的分支进行优化的结果，而 break 的存在也可以看成是由以前的汇编语言编程派生而来。

于是，在汇编语言已经十分罕见的现在，这种语法成为一个“谜”也是没有办法的事，但在 Go 中就没有这样的制约了。

最后，Go 还有一种独有的，很有趣的语法，那就是 switch 语句中判断分支条件的表达式是可以省略的（图 6）。这其实可以看成是用一种更易读的方式来实现一个 if-else 结构，实际上编译出来的结果貌似也是一样的。

```
switch {
 case a < b: return -1;
 case a == b: return 0;
 case a > b: return 1;
}
```

图 6　分支条件表达式可以省略

有趣的是，Ruby 的 case 结构也可以写成同样的形式，用 Ruby 编写出来的程序如图 7 所示。那个，我并不是说 Go 是抄袭 Ruby 哦，没有证据证明这一点，而且我也觉得不大可能会抄 Ruby，不过说实话，我还真小小地动过一点这个念头，万一是真的呢？（笑）

```
case
  when a < b: return -1
  when a == b: return 0
  when a > b: return 1
end
```

图 7　Ruby 中也可以省略分支条件表达式

for 结构也和 C 语言非常相似。和 if 结构一样，条件表达式不需要用括号括起来，但循环体则必须用花括号括起来，除此之外，还有一些地方和 C 语言有所不同。

首先，Go 的 for 语句中，条件部分的表达式有两种形式：单表达式形式和三表达式形式。其中单表达式形式和 C 语言中的 while 语句功能是相同的。

```
for 条件 {循环体}
```

三表达式形式则和 C 语言的 for 语句是相同的。

```
for 初始化; 条件; 更新 {循环体}
```

因此，编写出来的循环代码和 C 语言几乎是一样的：

```
for i=0; i<10; i++ {
  ...
}
```

有趣的是，在 Go 中，++ 递增并不是表达式，而是作为语句来处理的。此外，由于没有类似 C 语言中逗号操作符的形式，因此 for 的条件部分无法并列多个表达式。如果要对多个变量进行初始化，可以使用多重赋值：

```
for i,j=0,1; i<10; i++ {
  ...
}
```

在 for 语句中，空白表达式表示真，因此：

```
for ;; {
  ...
}
```

就表示无限循环，在 C 语言中也是一样的。

循环可以通过 break 和 continue 来中断，这两个语句的意思和 C 语言是一样的，不过 Go 中可以通过标签来指定到底要跳出哪一个循环（图 8）。

这一点又有点像 Java 的风格了，看来 Go 的设计真是研究和参考了其他多种语言呢。

```
Loop: for i = 0; i < 10; i++ {
  switch f(i) {
    case 0, 1, 2: break Loop
  }
  g(i)
}
```

图 8　用标签来指定要跳出的循环

作为控制结构来说，还有一些像 go 语句这样与并发编程关系密切的方式，我们稍后再来详细介绍。

类型声明

正如我们刚才所举的这些例子，Go 是一种深受 C 语言（不是 C++）影响的语言。不过，Go 也有和其他一些 C 派生语言不同的地方，其中最具有特色的就是其类型声明了。简单来说，Go 的类型声明和 C 语言是“正好相反”的。

C 语言中，类型声明的基本方式是：

```
类型　变量名;
```

不过，由于存在用户自定义类型，因此当遇到某个“名称”开头的一行代码时，很难一眼就判断出来这到底是类型声明，还是函数调用，或者是变量引用。

对人类来说存在歧义的表达方式，对编译器来说也就意味着需要更复杂的处理才能区分。因此，Go 中规定：声明必须以保留字开头，且类型位于变量名之后。

根据这两条规则，Go 的声明如图 9 所示。从深受 C 语言影响这个印象来说，这还真是令人震惊，不过习惯之后也就不觉得那么特殊了。这样的描述方式可以减少歧义，无论是对人还是

编译器来说都更加友好。

“:=”赋值也颇具魅力。“:=”赋值语句表示在赋值的同时将左侧的变量声明为右侧表达式的类型。

采用静态类型的语言中，由于需要大量对类型的描述，因此程序会通常会显得比较冗长，但在 Go 中由于可以省略类型声明，因此可以让程序变得更加简洁。虽说如此，但这种类型推导并非像某种函数型语言一样完美，因此 Go 也并非完全不需要类型声明。

Go 中没有 C++ 的模板（Template），也没有 Java 的泛型（Generic），但仅靠内置的数组（Array）、切片（Slice）、字典（Map）、通道（Channel）等类型，就可以指定其他的类型。切片是 Go 特有的一种类型，粗略来说，也可以理解成是数组的指针；字典则类似 Ruby 中的 Hash；通道用于并发编程，因此稍后再进行介绍。

```
// 字符串数组
var a [5]string

// 字符串切片
var s []string

// int到字符串的字典
var m map [int]string
// int管道
var c chan int
```

```
// 新类型的声明
type T struct {
  x, y int
}

// 常量的声明
const N = 1024

// 变量的声明
var t1 T = new（T）

// 变量声明的简略形
t2 := new（T）

// 还可以声明指针
var t3 *T = &t2

// 函数声明
func f（i int）float {
  ...
}

// 函数声明(多个返回值)
func f2(i int)（float, int）{
  ...
}
```

图 9　Go 的声明

目前，由于 Go 没有支持泛型，因此无法定义类型安全的用户自定义集合。

要取出用户自定义集合的元素，需要使用 Cast。

Cast 的语法如下：

```
f := stack.get.（float）
```

Cast 在执行时会进行类型检查，这让人不禁想起支持泛型之前的早期 Java 呢。

在 Go 的 FAQ 中，并没有否定将增加泛型的可能性，只不过优先级比较低，此外，随着对泛型的支持，类型系统会变得非常复杂，从这些因素来考虑的话，暂时还没有支持泛型的计划。

不过，个人推测既然早晚要支持泛型，那么现在是不是应该让现有的复合类型（数组、切片、字典等）的声明更具有统一性呢？

无继承式面向对象

了解了 Go 语言之后，从个人观点来看，我感触最深的，莫过于其面向对象功能了。Go 虽然是一种静态语言，但却拥有用起来感觉和动态语言相近的面向对象功能。

这其中最大的特征就是无继承，但它也不是基于原型（Prototype）那样的实现方式。Go 的面向对象机制和其他语言大相径庭，所以一开始很容易搞得一头雾水。

首先，Go 中几乎所有的值都是对象，而对象就可以定义方法。Go 的方法是一种"指定了接收器（Receiver）的函数"，具体如图 10 所示。

```
func (p *Point) Move(x, y float) {
  ...
}
```

图 10　Go 的方法定义中指定了接收器

函数名（Move）前面用括号括起来的部分"p *Point"就是接收器。接收器的名称也必须逐一指定，这一点挺麻烦的，不由得让人想到 Python。

有趣的是，方法的定义和类型的定义可以在完全不同的地方进行。这有点像 C# 中的扩展方法，即可以向现有类型中添加新的方法。

貌似像 int 这样的内置类型不能直接添加方法，不过我们可以给它起个别名叫 init，然后再向这个类型添加方法。

方法的调用方式还是比较普通的：

```
p.Move(100.0, 100.0)
```

和 C 语言不同，语言本身可以区分是否为指针，因此不需要自己判断是用"."还是"->"。

由于 Go 没有继承，因此通常的变量没有多态性，方法调用的连接是静态的。换一种更加易懂的说法，也就是说，如果变量 p 是 Point 型的话，则 p.Move 必定表示调用 Point 型中的 Move 方法。

然而，如果只有静态连接的话，作为面向对象编程语言来说就缺少了一个重要的功能。因此在 Go 中，通过使用接口（Interface），就实现了动态连接。

Go 的接口和 Java 的接口相似，是不具备实现的方法的集合，具体定义如下：

```
type Writer interface {
  Write(p []byte) int
}
```

interface 正文中出现的只可能是方法的类型声明,因此不需要保留字 func 和接收器类型。“不写不需要的东西”正是 Go 的风格。

作为未实现的类(类型),接口中只定义了方法的类型,而它本身也是一个类型,因此可以用于变量和参数的声明中。于是,只有通过接口来调用方法时,才会进行动态连接。

虽然语法有些差异,但大体上和 Java 的接口还是非常相似的。由于没有继承,因此只能通过接口来实现动态连接,这样便增加了静态链接的几率,提升了运行效率,这一点很有意思,不过也没有什么更大的好处了。

Go 的接口中令人感到惊讶的一点,就是某个类型对于是否满足某个接口,不需要事先进行声明。

在 Java 中,如果某个类在定义时用 implements 子句对接口进行了声明,则表示该类一定满足这个接口。然而,在 Go 中,无论任何类型,只要是接口中定义的方法(群)所拥有的类型,就都能满足该接口。

以上述 Writer 接口为例,只要一个对象拥有接受 byte 切片的 Write 方法,就可以进行代入。通过这个变量来调用方法的话,就会根据对象选择合适的方法来进行调用。

这不就是动态语言所推崇的鸭子类型吗?明明是一种静态语言,却如此轻易地实现了鸭子类型,让人情何以堪。

例如,我们前面经常提到的 fmt.Printf 方法,它的参数应该具有 String() 方法。

反过来说,只要对 String() 方法进行重新定义,就可以控制 fmt.Printf 方法的输出。其实,在我上学的时候,曾经对静态类型的面向对象语言十分着迷,也曾经模模糊糊地设想过类似这样的一个机制,但当时的我还没有能力将它实现出来。

Go 所提供的面向对象功能十分简洁,但却兼具了类型检查和鸭子类型(虽然当时还没有这么一个专有名词)两者的优点,这是何等优秀的设计啊!我非常感动。

那么,动态连接就通过接口这一形式实现了。然而,接口却无法实现继承所具备的另一项功能,即“实现的共享”。在 Java 中,即便使用接口也无法共享实现,因此大家普遍使用结合体(composite)这一技术。

对于这一点，Go 也考虑到了。在 Go 中，如果将结构体的成员指定为一个匿名类型，则该类型就被嵌入到结构体中。在这里很重要的一点是，嵌入的类型中所拥有的成员和方法也被一并包含到结构体中，事实上这相当于是多重继承呢。这样一来，大家可能会想，成员和方法的名称会不会发生重复呢？ Go 是通过下列这些独特的规则来解决这一问题的：

- 当重复的名称位于不同层级时，外层优先
- 当位于相同层级时，名称重复并不会引发错误
- 只有当拥有重复名称的成员被访问时，才会出错
- 访问名称重复的成员时，需要显式指定嵌入的类型名称

最后一条规则好像不是很容易看懂，我们来看一个示例（图 11）。

在图 11 中，结构体 C 和嵌入在其中的结构体 B 都拥有 z 这一名称重复的成员。然而，由于 z 位于外层，因此是优先的。如果要访问在 B 中定义的 z，则需要使用 B.z 这样的名称。

结构体 A 和结构体 B 都拥有 y 这一名称重复的成员。因此，包含 A 和 B 两个嵌入类型的结构体 C 中，两个重复的 y 成员位于同一层级。

```
type A struct {
  x, y int
}
type B struct {
  y, z int
}
type C struct {
  A           // x, y
  B           // y, z --y与A重复
  z int       // z与B重复
}
```

图 11　重复时的优先级示例

于是，当引用结构体 C 的 y 成员时，就会出错。在这种情况下，就需要显式指定结构体的名称，如 A.y、B.y，这样来访问成员才不会出错。这种设计真是相当巧妙。

多值与多重赋值

如前所述，Go 的函数和方法是可以返回多个值（即多值）的。

返回一个值需要使用 return 语句，但如果在声明返回值时指定了变量名，则可以自动在遇到 return 语句时返回该指定变量的当前值，而不必在 return 语句中再指定返回值（图 12）。

```
// 函数定义(多个返回值)
func f3(i int)(r float, i int){
  r = 10.0;
  i = i;
  return; // 返回10.0和i
}
```

图 12　return 返回 r 和 i

接受返回值采用的是多重赋值的方法：

```
a, b := f3(4); // a=10.0; b=4
```

Ruby 也可以通过返回数组的方式实现和多值返回类似的功能，但返回数组说到底依然只是返回了一个值而已，而 Go 是真正返回多个值，在这一点上做得更加彻底。

Go 的错误处理也使用了多值机制。相比之下，C 语言由于只能返回单值，且又不具备异常机制，因此当发生错误时，需要返回一个特殊值（如 NULL 或者负值等）来将错误信息传达给调用方。

UNIX 的系统调用（system call）和库调用（library call）大体上也采用了类似的规则。然而，在这样的规则下，正常值也可能和错误值发生重复，因此总有碰钉子的时候。

例如，UNIX 的 mktime 函数，在正常时返回从 1970 年 1 月 1 日 00:00:00UTC 开始到指定时间所经过的秒数，而出错时则返回 -1。然而在最近的系统平台中，也开始支持负值的秒数了，即 -1 变成了一个正常值，代表 1969 年 12 月 31 日 23:59:59。

Go 也没有异常处理机制。但通过多值，就可以在原本返回值的基础上，同时返回错误信息值，这被称为“逗号 OK”形式。

例如，在 Go 中打开文件的程序如图 13 所示。

```
f,ok := os.Open（文件名,os.O_RDONLY,0）;
if ok != nil {
 ... open失败时的处理...
}
```

图 13　文件的打开

这样的程序中，错误值和正常值可能会发生混淆。

和泛型一样，异常处理也是一个被搁置的功能，理由是会让语言变得过于复杂。不过，有了“逗号 OK”形式，在一定程度上就可以弥补缺少异常处理的不足。

然而，没有异常处理，也有不方便的地方。就是 Java 的 finally，或者 Ruby 的 ensure 的部分，即无论是正常结束还是发生异常，都要保证执行的后处理程序。

在 Go 中，是通过 defer 语句来实现后处理的。defer 语句所指定的方法，在该函数执行完毕时一定会被调用。

例如，为了保证打开的文件最终被关闭，可以像图 14 这样使用 defer 语句来实现。

```
f,ok := os.Open（文件名,O_RDONLY,0）;
defer f.Close（）;
```

图 14　使用 defer 关闭文件

在 Ruby 中，open 方法是完全不需要进行 close 的，而 Go 的抽象度虽然不如 Ruby 那样高，但也提供了可以避免文件忘记被关闭所需要的基本框架。

并发编程

如果要举出 Go 作为最新的系统编程语言最重要的一个特征，恐怕大多数人都会说——并发编程。

近年来，虽然像 Erlang 这些以并发编程为卖点的编程语言受到了广泛的关注，但在系统编程领域还没有出现这样的语言。要在系统编程领域实现并发编程，只能用 C、C++ 和线程（thread）做艰苦卓绝的斗争。

然而，线程这东西，在并发编程上绝对算不上是好用的工具。

Go 则在语言中内置了对并发编程的支持，这一功能参考了 CSP（Communicating Sequential Processes，通信顺序进程）[①]模型。

具体的方法是使用 go 语句。go 是 Go 特有的一个语句，也许这才是 Go 这个命名的来源吧。通过这个语句，可以创建新的控制流程。

```
go f（42）;
```

f 函数在独立的控制流程中执行，和 go 语句后面的程序是并行运作的。

这里所说的独立的控制流程被称为 goroutine，而控制流程还有其他一些表现方式，表 1 对它们的差异进行了比较。

表1 “控制流程”的实现方法一览

表现控制流程的术语	内存空间共享	轻　　型	上下文切换
process（OS）	no	no	自动
process（Erlang）	no	yes	自动
thread	yes	no	自动
fiber/coroutine	yes	yes	手动
goroutine	yes	yes	自动

其中，内存空间共享指的是某个控制流程是否可以访问其他控制流程的内存状态。

如果不共享的话，就可以避免如数据访问冲突等并发编程所特有的难题。但另一方面，为了共享信息，则需要对数据进行复制，但这样存在性能下降的风险。

① CSP（通信顺序进程）是一种用来描述并行系统交互模式的形式语言，最早由托尼 • 霍尔（C.A.R.Hoare, 1934—　）在其 1979 年的一篇论文中提出。CSP 对并发编程语言的设计有深远影响，受其影响的编程语言包括 Limbo 和 Go 等。

“轻型”是指一个程序是否可以创建大量的控制流程。例如，操作系统提供了进程（process）和线程（thread），但由一个程序创建上千个进程或线程是不现实的。然而，如果是轻型控制流程的话，不要说上千个，某些情况下就是创建上百万个也毫无问题。

最后，“上下文切换”是指在程序中是否需要显式地对控制流程进行切换。例如 fiber（也叫 coroutine）就需要进行显式切换。

除此之外，可以在等待输入暂停运行时，或者以一定时间间隔的方式自动进行切换。此外，支持自动上下文切换的方式（在大多数情况下）都支持在多核 CPU 中的多个核心上同时运行。

Go 的 goroutine 支持内存空间共享，是轻型的，且支持自动上下文切换，因此可以充分利用多核的性能。在实现上，是根据核心数量，自动生成操作系统的线程，并为 goroutine 的运行进行适当的分配。

此外，通过使用自动增长栈的 segmented stack 技术，可以避免栈空间的浪费，即便生成大量的 goroutine 也不会对操作系统带来过大的负荷。

在内存空间共享的并发编程中，如果同时对同样的数据进行改写就会发生冲突，最坏的情况下会导致数据损坏，甚至程序崩溃，因此必须要引起充分的注意。在 Go 中，为了降低发生问题的几率，采取了下面两种策略。

第一种策略是，作为 goroutine 启动的函数，不推荐使用带指针的传址参数，而是推荐使用传值参数。这样一来，可以避免因共享数据而产生的访问冲突。

第二种策略是，利用通道（channel）作为 goroutine 之间的通信手段。通过使用通道，就基本上不必考虑互斥锁等问题，且以通道为通信方式的并发编程，其有效性已经通过 Erlang 等语言得到了证实。

通道是一种类似队列的机制，只能从一侧写入，从另一侧读出。写入和读出操作使用 <– 操作符来完成（图 15）。

```
// 创建通道
 c := make(chan int);
 c <- 42      // 向通道添加值
 v := <- c    // 取出值并赋给变量
```

图 15　<– 操作符的使用示例

图 16 是一个用 goroutine 和通道编写的简单的示例程序。这个程序中，通过通道将多个 goroutine 连接起来，这些 goroutine 分别将值加 1，并传递给下一个 goroutine。

向最开始的通道写入 0，则返回由 goroutine 链所生成的 goroutine 个数。这个程序中我们生成了 10 万个 goroutine。

在我的配备 Core2 Duo T7500 2.20GHz CPU 的电脑上运行这个程序，只需要不到 1 秒的时间就完成了。生成 10 万个控制流只用了这么短的时间，还是相当不错的。

```
package main
import "fmt"

const ngoroutine = 100000
func f(left, right chan int) { left <- 1 + <-right }
func main() {
   leftmost := make(chan int);
   var left, right chan int = nil, leftmost;
   for i := 0; i < ngoroutine; i++ {
     left, right = right, make(chan int);
     go f(left, right);
   }
   right <- 0; // bang!
   x := <-leftmost; // wait for completion
   fmt.Println(x);   // 100000
}
```

图 16　Go 编写的并行计算示例程序

小结

Go 是一种比较简洁的语言，但我们在这里依然无法网罗其全部方面。因此，我在这里对 Go 的介绍，是从某种语言的设计者在看待另一种编程语言的时候，会被哪些点所吸引这个角度出发的。

我认为 Go 是一种考虑十分周全的语言。我做了很多年的 C 程序员（还有一段做 C++ 程序员的历史）。作为一种系统编程语言，让我稍许产生“也许可以换它用用看”这样念头的，从上学时用上 C 语言以来到现在，这还是头一次。

当然，Go 也不是一种十全十美的语言，它也有诸多不足。如数组、字典等特殊对待的部分，以及作为一种静态语言，总归还是需要对泛型做出一定的支持等。

此外，异常处理也是很有必要的。即便有可能会让语言变得复杂，我认为最好还是应该加上对方法重载和操作符重载的支持。而对于是否可以省略分号这样的规则，对我来说并没有什么直观的感受。

在实现方面，Go 的目标是做到压倒性的高速编译，以及将运行所需时间控制在比同等 C 语言程序多 10% 到 20% 的范围内。但就目前来看，先不要说编译时间，貌似连运行时间也尚未达到当初的目标。

不过，回过头来想想，Go 还只是一种非常年轻的语言。从 2007 年项目启动算起，也只是仅仅过了几年的时间而已。

一种编程语言从出现到被广泛关注和使用，大多都需要 10 年以上的时间，而 Go 只用了短短几年的时间就走到这一步，着实令人惊叹。

Go 是下一代编程语言中我最看好的一个，今后也会继续关注它的发展。

说句题外话，其实在 Go 出现很久之前，就已经存在一种叫做“Go!”的语言了。由于 Google 奉行“不作恶”（Don’t be evil）的信条，因此网上有很多人认为 Go 应该改名。

话说，语言名称撞车也不是什么新鲜事（用 Ruby 这个名字的编程语言也有好几个），不过网上有人推荐将 Go 语言改成 Golang 或者 Issue-9。前者是来自 Go 官方网站的域名（golang.org），后者则是来自“已经有一个叫 Go! 的语言了，请改名”这个问题报告的编号[①]。

就我个人来说，我会给“不改名，撞车就撞车”这个选项投上一票。如果非要改的话，我比较喜欢 Golang 吧。无论如何，我对 Google 今后会做出怎样的抉择十分关注。

① Issue-9 来源于 Google Code 中 Go 语言 Issue tracker 中的第 9 号问题。该问题提交于 2009 年 11 月 10 日（即 Go 刚刚发布几天之后），标题为“I have already used the name for *MY* programming language”（我已经为 * 我自己的 * 编程语言用了这个名字），发帖人称他自己一个已经开发了 10 年的语言也叫这个名字（有人指出其实发帖人开发的语言叫“Go!”，多一个感叹号），并且发表了论文还出版了书。2010 年 10 月，Google 正式作出回应，称叫 Go 的计算机相关产物有很多，过去 11 个月中这两种语言命名的类似性也没有引起很大的歧义，因此决定关闭该问题。（Issue-9 原帖链接：http://code.google.com/p/go/issues/detail?id=9）

3.3 Dart

2011 年 10 月在丹麦奥胡斯市召开的 GOTO 大会 2011 上，Google 公司发布了一种新的编程语言 Dart。

GOTO 大会每年都在奥胡斯市召开，这个活动曾经叫做 JAOO（Java and Object Oriented，Java 与面向对象），在欧洲算是首屈一指的技术大会。《代码重构》的作者马丁・福勒（Martin Fowler）、维基创始人沃德・坎宁安（Ward Cunningham）[①]、“编程达人”大卫・托马斯（Dave Thomas）、C++ 创始人比雅尼・斯特劳斯特鲁普（Bjarne Stronstrup）[②]等著名的技术先驱都曾经作为演讲者在该大会上发表过演讲。

我自己也有两次登台演讲的经历，其中一次是在 2001 年。那个时候 Ruby on Rails 还没有诞生，可以说主办方的眼光十分敏锐。所有的演讲者都称赞大会的讲师阵容豪华、料理好吃，堪称“最棒的大会”。

其实，David Heinemeier Hansson[③]也曾作为学生工作人员参加了 2001 年那次大会。传说，他是借在会后的饭局上跟我聊天的机会，对 Ruby 产生了兴趣，从而从 PHP 转到了 Ruby，之后在美国 37signals 公司开发出了 Ruby on Rails。

关于 JAOO 的题外话好像有点太多了。虽说对我个人来说这个大会给我留下了很深的印象，不过这个话题还是到此为止吧。下面我们回到主题，来讲讲 Dart。

为什么要推出 Dart？

像“Dart 语言入门”这样的题材，不如还是留给别的杂志、图书和网站来做吧，在本书中，我们的介绍重点关注的是隐藏在 Dart 背后的“为什么”。当然，Google 公司并没有官方公布过

① 沃德 • 坎宁安（Ward Cunningham，1949—　）是一位美国计算机程序员，维基（Wiki）概念的发明者。
② 比雅尼 • 斯特劳斯特鲁普（Bjarne Stroustrup，1950—　）是一位计算机科学家，C++ 的创始人，现任德克萨斯州 A&M 大学工程学院计算机科学首席教授。
③ David Heinemeier Hansson（1979—　）是一位丹麦计算机程序员，Ruby on Rails 的创始人，在 Ruby 社区中常用名为 DHH。

推出 Dart 的意图，我也只是从声明以及语言设计规格中推测的。不过，即便是以这些有限的信息为出发点，却也得到了很多意外的收获。

那么，Google 公司到底为什么要开发和发布一种新的编程语言呢？像 Ruby 这样由一个人开始开发的语言，仅仅拥有对技术的兴趣，以“想做做看而已”这样的理由就足够成立了。但是 Google 公司作为一家世界上具有代表性的企业，用自己公司的名义来发布一种新的编程语言，我觉得其中一定另有深意。

况且，很多人都知道，在 Google 公司中有这样一条规定，公司内部的软件开发项目，只能使用 C/C++、Java、Python 和 JavaScript 这几种语言。之所以有这条规定，是因为所使用的语言种类越多，就需要雇佣越多精通这些语言的技术人员，而限制开发语言的种类，主要是从降低管理成本上来考虑的。软件开发是 Google 公司的生命线，先不站在技术人员兴趣的角度上来考虑，就从维系这一生命线需要管理大量的代码这个角度来看，毋庸置疑这是在公司经营层面上一个非常合理的判断。因此，虽然编程语言的开发在技术上非常吸引人，但 Google 也决不会草率地在自己公司里开发一种编程语言并发布出来。Google 不惜违背自己的规定而开发一种自己的编程语言，这背后到底有怎样的原因呢？

2009 年 Go 发布的时候，官方对于动机的解释是为了克服 C/C++ 的缺点。也就是说，Google 公司要开发的软件数量实在是太庞大了，像 C 语言这样设计古老的语言（诞生于 1972 年）便遇到了瓶颈，而 C++ 的设计由于考虑了和 C 语言之间的兼容性，因此也显得有些力不从心了。

因此，Google 公司的开发人员希望能够提供一种：

- 更现代
- 更安全
- 十分高速

的替代语言。的确，Go 在语言设计上保持了和 C 语言同等程度的高速性，同时还加入了简洁的面向对象功能、垃圾回收机制以及类型安全等特性。

进一步推测的话，像 Google 这样需要编写大量代码的公司，即便没有什么外部用户，光公司内部应该也可以保证足够多的使用者。虽然 Go 也曾在一段时间内并没有引起太大关注，但对 Google 公司来说应该也算不上是什么问题吧。

应该说，Dart 背后应该也隐藏着类似的动机，当然在这里需要被替换的语言成了 JavaScript。JavaScript 是美国原 Netscape Communications 公司作为其浏览器产品的内部语言而开发的，开发周期非常短。

JavaScript 的设计者布兰登·艾克（Brendan Eich）[①]曾在一次采访中说，JavaScript“几天就设计出来了”。从这样的出身来看，出乎意料，它还真算是一种做得不错的语言。但由于开发周期短，确实也存在着诸多不足。例如，开发周期短导致了语言规格和实现过于追求简单化，而程序员实际开发出的 JavaScript 代码则容易变得十分繁杂。

在 Dart 发布前夕，曾经从 Google 公司内部泄露出了一份备忘录，内容如下[②]：

- ❑ JavaScript 包含一些语言本质上的缺陷，这些缺陷无法通过对语言的改进来解决。因此，我们将对 JavaScript 的未来执行两个方面的战略。
- ❑ Harmony（低风险、低回报）：与 ECMAScript 标准规范小组 TC39 合作，继续努力对 JavaScript 进行改进。
- ❑ Dash[③]（高风险、高回报）：在保持 JavaScript 动态性质的同时，开发一种更容易提升性能的、更容易开发适合大规模项目所需工具的新语言——Dash。

关于执行这两方面战略的理由，这份备忘录给出了如下解释。首先，如果拘泥于 JavaScript，那么 Web 的发展就会发生停滞，从而就可能在与苹果公司的 iOS 等对手的竞争中失利。但反过来说，如果放弃 JavaScript 而只专注于 Dart，一旦 Dart 失败，则 JavaScript 的发展就会停滞，最坏的情况下甚至会危及 Google 公司在技术界的地位。因此，Google 才做出了这种“两手抓、两手都要硬”的决定。

对于这份备忘录，JavaScript 阵营，尤其是该语言的创始人布兰登·艾克回应说，性能和工具支持都不是什么大问题，即便是现在的 JavaScript 有一些缺陷，也是可以进行改善的。而且 JavaScript 的下一个版本 Harmony 中，已经对这些问题进行了一定程度的应对。

此外，JavaScript 目前受到的主要批判，如：

- ❑ 无法应对复杂的互联网应用程序
- ❑ 无法进行高速化
- ❑ 不支持多核 /GPU
- ❑ 无法修正

但这里面存在着一些误解，因此他们主张，今后还是应该专注于 JavaScript。而且，开发一种新的语言，可能会造成社区的分裂。

---

① 布兰登·艾克（Brendan Eich，1961—　）是一位计算机程序员，JavaScript 语言的创始人，现任 Mozilla 公司首席技术官（CTO）。

② Google 公司对于这份备忘录的真伪没有做出官方表态。（原书注）

③ 在这份备忘录中，Dart 被称为 Dash，应该是项目途中更改了名称。（原书注）

技术的正确与否，只能留给将来的历史去证明，我们在这里不去判断双方孰优孰劣，但至少我认为，他们双方的主张都具备各自的合理性。

下面，我们就来具体看一看 Dart 这个语言吧。

Dart 的设计目标

在 Dart 的主页 dartlang.org①中，关于 Dart 的设计目标是这样说明的：

- ❑ 创造一种结构化而又十分灵活的 Web 开发语言。
- ❑ 要让 Dart 对程序员更加自然和友好，作为结果，将 Dart 设计成一种容易学习的语言。
- ❑ 构成 Dart 的全部语言机制都不应该对高速运行和快速启动产生妨碍。
- ❑ 要将 Dart 设计成一种能够适应一切 Web 相关设备的语言，包括手机、平板电脑、笔记本电脑、服务器等。
- ❑ 要在主流的现代浏览器上提供可以高速运行 Dart 的工具。

需要实现上述目标的 Web 开发者，所遇到的问题有下面这些：

- ❑ 小型脚本通常在没有实现结构化的情况下就成为了一个大型的 Web 应用程序，这样的应用程序很难进行调试和维护。而且，这种一整块的应用程序，无法由多个团队分担开发工作。Web 应用一旦变得巨大，就无法保证其开发效率。
- ❑ 能够快速编写代码的轻量化特性，是脚本语言受欢迎的原因。这样的语言中，一般来说，对应用程序模块间访问的约定（契约），并不是由语言本身来完成的，而是通过注释来表现的。结果，除了作者以外，要读懂代码并进行维护就变得非常困难。
- ❑ 现存的语言中，都要求开发者必须从静态类型和动态类型两者中选择一种。传统的静态类型语言都需要庞大的开发工具，编码风格上的制约也比较多，让人感觉缺少灵活性。
- ❑ 开发者无法在服务器和客户端上构建一个具备统一感的系统。node.js 和 Google Web Toolkit（GWT）是为数不多的例外。
- ❑ 多种语言和格式的混合，会导致繁杂的“上下文切换”问题，增加了开发的复杂性。

原来如此。作为动态语言的信奉者，我无法完全同意这些观点，不过我想就算我不说，大家也应该能理解的吧。那么，既然意识到这些问题的存在，为了实现所设定的目标，Google 公司又将 Dart 设计成了怎样一种语言呢？

① 将“语言名称 +lang”作为官方网站域名，最早应该是我为 Ruby 创建的“ruby-lang.org”。虽然没有证据证明 Google 是受了我的影响，但想象一下我的想法居然能够影响到 Google，就感觉到一种莫名的开心。（原书注）

代码示例

首先，我们来看一段 dartlang.org 上面的示例程序（图 1）。这个，嗯，怎么说呢，感觉和 Java 很像啊。

```
interface Shape {
 num perimeter();
}

class Rectangle implements Shape {
 final num height, width;
 Rectangle(num this.height, num this.
width);
 num perimeter() => 2*height + 2*width;
}

class Square extends Rectangle {
 Square(num size): super(size, size);
}
```

图 1　Dart 示例程序（1）

不过，仔细看看就会发现还是有很多不同的。例如数值类型叫做 num，还有构造方法的编写十分简洁，方法定义有其他形式等。此外，super 的用法和 Java 也有些区别，不指定方法名这样的形式又有点像 Ruby。

我们再来看另外一段示例程序（图 2）。这次好像风格变得有点不一样了。

```
main() {
 var name = 'World';
 print('Hello, ${name}!');
}
```

图 2　Dart 示例程序（2）

怎么样？是不是感觉和 Java 有点相似，但又比 Java 要简洁？说起和 Java 语法相似的脚本语言，让我想起了 Groovy[①]。Dart 作为 JavaScript 的后继者，试图对简洁的编程提供支持，大家是否从中感觉到了呢？用 “$” 将表达式嵌入到字符串中，这一点倒是很有脚本语言的风格。Ruby 也差不多，只不过用的是“#”而不是“$”。哦对了，Groovy 也是用“$”在字符串中嵌入表达式的。

Dart 中无需显式指定类型，程序以 main 方法作为起点。Dart 最大的特征就在于其类型声明是可以省略的。关于这种 “非强制性静态类型”的机制，我们稍后会详细进行探讨。

```
class Greeter {
 var prefix = 'Hello,';

 greet(name) {
   print('$prefix $name');
 }
}

main() {
 var greeter = new Greeter();
 greeter.greet("Class!");
}
```

图 3　Dart 示例程序（3）

下面我们来创建一个类（图 3）。这次又很像 Groovy 的风格呢。这个程序很简单，就是创建一个类，并调用它的方法，好像没有什么讲解的必要呢。

① Groovy 是一种 Java 平台上的面向对象编程语言，发布于 2003 年。它是一种动态语言，可以作为 Java 平台上的脚本语言来使用。

Dart 中可以创建多个构造方法。那么，我们来定义一个指定问候词的构造方法吧（图 4）。深受 Ruby 毒害的人肯定会说，这种功能用类方法来实现不就好了嘛。

Dart 的实例变量默认是公有（public）的，可以从外部进行访问。因此：

```
greet.prefix = "Goodbye"
```

就可以改写实例变量了。如果不希望公开实例变量的话，就需要将实例变量声明为私有（private）。此外，为了对属性访问进行抽象化，还可以定义 setter 和 getter 方法。如果将图 3 的程序修改一下，将 prefix 私有化并用 setter 和 getter 进行封装，就变成了图 5 的样子。

```
class Greeter {
 var prefix = 'Hello,';

 Greeter();
 Greeter.withPrefix(this.
prefix);
 greet(name) {
   print('$prefix $name');
 }
}

main() {
 var greeter = new Greeter
   .withPrefix('Howdy,');
 greeter.greet("Class!");
}
```

图 4　Dart 示例程序（4）

```
class Greeter {
 String _prefix = 'Hello,';          // Hidden instance variable.
 String get prefix() => _prefix;     // Getter for prefix.
 void set prefix(String value) {     // Setter for prefix.
   if (value == null) value = "";
   if (value.length > 20) throw 'Prefix too long!';
   _prefix = value;
 }

 greet(name) {
   print('$prefix $name');
 }
}

main() {
 var greeter = new Greeter();
 greeter.prefix = 'Howdy,';          // Set prefix.
 greeter.greet('setter!');
}
```

图 5　Dart 示例程序（5）

首先，名字以“_”开头的变量是私有的。私有的实例变量无法从外部进行访问。setter 和 getter 是在方法名前面分别加上 set 和 get。在 Dart 中，setter/getter 和一般的方法是有明确区分的，因此无法定义和 setter/getter 重名的方法，此外，也无法在子类中重写这一类方法。

最后我们来简单说明一下泛型。拥有静态类型的语言，必然需要带参数的类型。因此，Dart 也理所当然地具有泛型。

在静态类型语言中，通过是否拥有带参数的类型，就能看出在语言设计的时候对于类型进

行了何种程度的考量。这让人想起，早期的 Java 和 C++ 都没有泛型和模板类呢。话说回来，考虑到那些语言出现的时间，一定程度上说，这也许是没办法的事吧。

在这里我想为 Java 平反一下。Java 其实在早期就探讨过引入带参数类型，但考虑到当时带参数类型还处于研究水平，恐怕很难在设计规格上达成一致。因此，在早期的规格中就放弃了这个功能。

```
class Greeter {
 var name;
 Greeter(this.name);
 greet() {
   print('Hello ${name}.');
 }
}

main() {
 List<Greeter> greeters = [new Greeter("you"),
new Greeter("me")];
 for (var g in greeters) g.greet();
}
```

图 6　Dart 的示例程序（6）

那么，作为现代的静态类型语言，Dart 也采用了泛型。我们在前面的示例中，尝试用了一下泛型，虽然有些牵强。在图 6 中，我们使用 List<Greeter> 来存放多个 Greeter 类。当然，由于在 Dart 中类型声明是非强制的，因此在这里用 var 程序也一样可以正常运行。

Dart 的特征

刚才我们对 Dart 进行了快速的了解。Dart（目前）并不是一种规格规模很大的语言，但以这点篇幅也不可能涵盖其全部特性，不过至少大家能对它的基本风格有所了解了吧。

因此，Dart 的特征，尤其是和 JavaScript 进行比较的话，我认为比较重要的应该是：

- 基于类的对象系统
- 非强制性静态类型

当然，除此之外还有其他一些细微的差异，但如果说 Dart 和 JavaScript 之间决定性的差异的话，我想非上述两点莫属了。

基于类的对象系统

在 JavaScript 中，对象的实现基本上是用散列表（hash table）的方式。JavaScript 中，除了数值和字符串，几乎所有的数据都是对象（散列表）或者函数（函数对象），也就是说，基本上是用这两种数据结构来“以不变应万变”。

散列表的数据取出访问数量级为 $O(1)$[①]，无论表的大小如何，都能以一定的速度来取出数据，是一种很优秀的数据结构。但遗憾的是，与直接访问数组和结构体相比，无论是数据的取出还是更新，所需的时间都要长得多。

在以 Google Chrome 内置的 v8 为代表的现代 JavaScript 引擎中，作为优化的一部分，在满足一定条件的情况下，会将对象以结构体的方式来实现。然而，Dart 天生就具备基于类的对象系统，因此不需要进行这种不自然的优化行为。作为结果，以简单的引擎来实现高性能，这一点是非常值得期待的。

话说，JavaScript 在下一个版本 Harmony 中也采用了基于类的对象系统。虽说这样一来，JavaScript 方面会面临着和原有版本的兼容性问题，但可以想象，今后 Dart 的优势将逐渐被削弱。

非强制性静态类型

Dart 最大的特征莫过于非强制性静态类型了。由于类型的描述和程序本身的逻辑没有直接关系，因此有很多人觉得类型是十分繁琐的，但类型也并非一无是处。首先，虽然类型信息与程序逻辑没有直接关系，但属于重要的附属信息。通过类型的矛盾，经常可以检查出程序的错误。虽说程序中的类型信息没有矛盾，这并不代表程序就没有错误，但至少有相当多的错误，是可以通过类型信息由机器检查出来的。

此外，通过在程序中附加类型信息，使得在编译时可以用来进行优化的信息增加，就更有可能生成出高品质和高性能的代码。进一步说，IDE 等工具的自动完成等辅助功能，也可以帮助更好地利用类型信息。

静态类型有如此多的好处，但另一方面，小规模的程序中如果强制对类型信息进行描述的话，类型信息所占的比例就会相当大，从而使得程序逻辑的本质被埋没，也会消磨开发的欲望。

为了解决这个矛盾，某些语言采用了类型推导（type inference）机制，而 Dart 则是采用了“非强制性（可省略）静态类型”（optional typing）的方法。在 Dart 中，没有指定类型的变量和表达式会被当做 Dynamic 型，其类型检查在运行时完成。

采用非强制性系统的语言并非只有 Dart，这些语言最大的问题在于，如果类型信息是非强制性（可省略）的，在运行过程中类型信息就会逐渐减少，导致可进行类型检查的范围不断缩小。结果，在编译时可以发现错误这一静态类型所具备的优势就没了一半。此外，随着类型信息的减少，能够用于优化的信息也同时减少，从这一点上来说也有点得不偿失。

① 在算法中，$O(N)$（大 O 记法）表示在最坏的情况下，一种算法的性能与其对象数据量之间的关系。$O(1)$ 表示算法性能与数据量无关，即数据量的大小不影响该算法的性能。更多关于大 O 记法的解释请参见 4.1 节中的相关内容。

基于这些问题，Dart 进行了大胆的突破。也就是说，Dart 是类似 JavaScript 这样，在语言本质层面不具备类型信息的动态语言，而静态类型信息仅仅是作为辅助地位而存在的。在 Dart 的语言规格中，明确记载了 Dart 具备完全不进行类型检查的工作模式。也就是说，在没有显式打开类型检查器的情况下，例如：

```
num n = "abc";
```

这样的程序是完全可以正常运行的。

大概很多人会问，这样到底有什么好处呢？说实话，我也有同样的疑问。

我就大胆推测一下，如果使用了带类型信息的库，IDE 等的自动完成功能已经十分有效，而且程序中也会进行一定程度的类型检查，这是其一。另外，随着自己所开发的程序规模逐渐扩大，可以阶段性地增加静态类型信息，从而同时享受了动态类型和静态类型双方的优点。

这样一说的话好像也能说得通，但与此同时，我还是会对这种机制是否能够成功表示怀疑。

Dart 的未来

那么，在这样的背景下诞生的 Dart，今后会不会普及呢？

个人认为，Dart 的未来还真不能说有多么光明。理由有很多，首先一个就是期望与现实的差距。

一种编程语言并不是有了语言的引擎就算完成了，而是必须在这种语言得以立足的库、框架、应用程序等“生态圈”成熟起来之后，其价值才真正开始体现。而要走到这一步，需要花上很多年的时间。Dart 诞生在 Google 公司这样的名门中，天生就被赋予了很大的期望，但要想实际建立起自己的生态圈，并成为一种可用的语言，所要花费的时间并不会和其他语言有什么不同。Dart 是否能够忍受住期望和现实之间的差距，目前还是未知数。

此外，Dart 当初的目标是为了打倒 JavaScript，但它的对手拥有大量的用户、社区和应用程序，作为新手的 Dart（尽管有 Google 公司作为后盾）却仿佛赤手空拳一般。基于类的对象系统也好，非强制性静态类型也好，虽然都是不错的概念，但这些是否具备足够的独创性和魅力，来弥补前面所说的压倒性劣势呢？我只能表示怀疑。还有，在 Dart 实用化之前，JavaScript 也一定会完成进一步的进化，战斗的形势十分严峻。

话说回来，编程语言是一种“10 年也就幼儿园小孩水平”的耐久型领域，未来的事谁都无法预测，我们只能继续关注 Dart 的发展了。

3.4 CoffeeScript

最近，JavaScript 的发展十分惊人，有一种语言试图借 JavaScript 之威风争得一席之地，下面我们就来介绍一下这种语言——CoffeeScript。

最普及的语言

世界上的编程语言种类相当多，据说有成千上万种。要说其中最普及的，或者换个说法，其引擎被安装数量最多的语言，恐怕非 JavaScript 莫属了。

最近的计算机用户都不大会去编程了，但几乎所有人都会访问网站吧。访问网站，甚至已经成为“上网”的代名词。现在世上几乎所有的 Web 浏览器都内置了 JavaScript 引擎。PC 上的 Internet Explorer、Firefox、Safari、Chrome 等自不必说，就连智能手机甚至是非智能手机的浏览器上都装上了 JavaScript 引擎。

随着移动设备的兴起，尤其是考虑到非智能手机的普及率，完全可以断言 JavaScript 就是最普及的语言。而正是因为有了如此之高的普及率，才进一步推动了其重要性的不断上升。

被误解最多的语言

另一方面，JavaScript 也可以说是被误解最多的语言。

JavaScript 是由原 Netscape Communications 公司的布兰登·艾克，于 1995 年开发的一种用于扩展浏览器功能的编程语言。最初它被命名为 LiveScript，但当时正好是美国 Sun Microsystems 公司（现被 Oracle 公司收购）的 Java 方兴未艾之际，再加上 Netscape 和 Sun 之间有业务上的合作，因此为了在市场宣传上更有冲击力，就改名为 JavaScript 了。JavaScript 中大量使用了花括号，看上去和 Java 有点像，但其语言核心意义的部分和 Java 是完全不同的，因此这个名字便成了招致重大误解的元凶。

在 JavaScript 还没成名的时候，就经常听到类似“JavaScript 就是 Java 吧”这样的说法，还有很多人认为只要学会 Java 也就学会了 JavaScript。

JavaScript 本来的目的，是为了编写点击网页按钮时所需的一些简单的处理逻辑，由这一点又招致了第二个误解——JavaScript 是只能完成简单工作的简易语言。然而实际上则出乎意料，JavaScript 是一种设计良好的语言，它拥有基于原型的面向对象功能，可以将函数作为对象来使用，在此基础上还提供了正式的闭包功能。由于它可以进行函数型编程，因此从语言的语义上来看，有接近 Scheme 的一面。

利用 JavaScript 的良好设计，微软公司实现了动态网页 DynamicHTML，像 Google 地图这样大量运用 JavaScript 的网站也开始不断出现，这让人们对于 JavaScript 的印象发生了转变。Google 地图是 Ajax（Asynchronous JavaScript and XML，异步 JavaScript 与 XML）编程风格的先驱。如今，使用 JavaScript 制作视觉特效，以及用 Ajax 实现无页面迁移的网站，已经一点都不稀奇了。

当初，JavaScript 作为 Netscape Navigator 浏览器内置的客户端语言问世，后来又逐渐内置到其他一些浏览器中。然而，由于各公司对 JavaScript 的实现是在参考 Netscape 的基础上独自开发的，因此浏览器之间的兼容性很低，这让程序员感到十分痛苦。早期的 JavaScript 程序员，需要运用各种各样的方法来判断浏览器类型，为了回避兼容性问题而做出很大的努力。在 1997 年，由 ECMA[①]规范作为“ECMAScript”实现标准化以来，这一问题得到了很大的改善。即便如此，依然还有一些人在使用着老版本的 Internet Explorer，因此大家还没有完全从兼容性问题中解放出来。

最后的误解是关于性能。JavaScript 的变量和表达式没有类型信息，具备动态性质，因此其性能和 Java 等静态类型语言相比具有劣势。实际上，早期的 JavaScript 引擎在实现上并没有过于追求性能，然而，随着 JavaScript 应用范围的扩大，这一点也得到了巨大的改善。

显著高速化的语言

作为编程语言来说，经常被关注的一点，就是同样的算法用各种不同的语言实现的时候，相互之间有多少性能上的差异。这一点上，一般认为采用能获得更多性能改善信息的静态类型，且以编译器作为引擎的语言性能比较高，例如 C++ 和 Java 等。

然而从根本上讲，性能应该与引擎的性质有关，而和语言的种类是无关的。因此，像

① ECMA 是一个信息和电信标准组织，原名“欧洲计算机制造商协会”（European Computer Manufacturer Assosications），后来随着该组织的国际化，虽沿用 ECMA 这个名称（一般称为 ECMA International），但已经不代表之前的全称了。该组织以 JavaScript 为基础制定的标准化规范“ECMAScript”编号为 ECMA-262，JavaScript 是该规范的兼容和扩展版本。

JavaScript 是动态语言因此速度慢这种印象并非普遍事实，而是由该语言的引擎在实现上做出了多大的努力而决定的。

的确，早期的 JavaScript 引擎性能并不算高。然而随着 JavaScript 被广泛使用，其重要性也跟着提高，对 JavaScript 引擎的投资也得到了扩大，各种高速引擎相继问世。刚刚诞生之际的 Java，由于需要通过字节码解释器来工作，和 C++ 等原生语言相比速度慢了不少，甚至有人说："这种东西完全不能用。"但仅仅过了不久，Java 的性能就得到了大幅度的改善，现在在某些情况下，甚至能够实现超越 C++ 等语言的性能，这和 JavaScript 现象十分类似。

最近的 JavaScript 引擎中，由于采用了 JIT、特殊化、分代垃圾回收等技术，在动态语言中已经可以归入速度最快的级别了。

JIT 是 Just In Time Compiler 的缩写，指的是在程序运行时将其编译为机器语言的技术。由于编译为机器语言的程序可以以 CPU 原本的速度来运行，因此能够克服解释器所带来的劣势。JIT 在 JVM（Java Virtual Machine，Java 虚拟机）中也得到了运用。

所谓特殊化，指的是一种在将函数转换为内部表达时所运用的技术。通过假定参数为特定类型，事先准备一个特殊化的高速版本，在函数调用的开头先执行类型检查，当前提条件成立时直接运行高速版本。动态语言运行速度慢的理由之一，就是因为在运行时需要伴随大量的类型检查，而通过特殊化则可以回避这一不利因素。

分代垃圾回收，是一种对不再使用的对象进行回收的垃圾回收（Garbage collection）算法。垃圾回收有一些比较普通的方法，如标记清除法。这种方法对由程序（变量等）引用的对象进行递归式扫描，标记出"存活对象"，并认为剩下的对象将来不再被访问，将其作为"死亡对象"进行回收。这种方法的缺点是，程序中生成的对象数量越多，为了找到存活对象所需的扫描次数就越多。如果运行时间的很大一部分都消耗在垃圾回收上的话，性能就会降低。JavaScript 开发的程序，随着规模的扩大，对象数量也跟着增加，采用标记清除法所带来的性能下降问题也就愈发显著。

要改善这个问题，其中一个方法就是分代回收。在大部分程序中都存在这样一种趋势，即所生成的对象的一大半都只被使用很短的一段时间就不再被访问了，而存活下来的一部分对象，却拥有非常长的寿命。在分代回收中，将对象划分为新生代和老生代（根据情况还可能划分更多的代）两个组。其中对新生代频繁进行扫描，而对老生代只偶尔进行扫描，从而减少了整体的扫描次数。

由于上述这些技术的运用，JavaScript 得以在为数不多的动态语言中跻身速度最快的行列。Ruby 当然也被超越了，感到相当寂寞呢。

对 JavaScript 的不满

那么，虽然 JavaScript 人气如此之高，使用又如此广泛，但随着用户数量的增加，还是招致了越来越多的不满。

JavaScript 从语法和语义上来看都非常简单，基本上是一种非常优秀的语言。然而，它的语法有些过于简单了，有很多人对程序容易变得冗长感到不满。多年以来，我一直主张过于简单的语言一定不会让程序员开心，因此这一不满也可以说是应验了我的观点吧。

为了让语言功能变得更加丰富，出现了一些如 prototype.js、jQuery 之类的库，其中增加了一些方法，让 JavaScript 的对象用起来有 Ruby 的感觉。当然，这些库所提供的功能并不仅限于此。

CoffeeScript

于是，为了解决对 JavaScript 语法上的不满，CoffeeScript 做出了尝试。CoffeeScript 是由 Jeremy Ashkenas 开发的。Ashkenas 拥有多种编程语言的经验，还开发过用于从 Ruby 访问视觉设计语言 Processing①的 Ruby Processing。

也许是出于这样的背景，CoffeeScript 在语法上貌似受 Ruby 和 Python 的影响很大。两者相比的话，应该还是受 Python 影响更大一些。

所谓 CoffeeScript，一言以蔽之，就是用 Javascript 实现的用于编写 JavaScript 的方便语言。CoffeeScript 是一种可以弥补 JavaScript 缺点和不满的、拥有独自语法的编程语言，和 JavaScript 之间完全没有兼容性。然而，CoffeeScript 程序在运行前需要被编译为 JavaScript，然后作为 JavaScript 程序来运行。也就是说，虽然程序看上去完全不同，但其语义的部分却是完全相同的。

进一步说，CoffeeScript 的编译器是用 JavaScript 编写的。也就是说，只要有 JavaScript，CoffeeScript 编写的程序就可以在浏览器上直接运行。很多语言都因为无法在客户端使用，从而不得不转向服务器端环境，而这一性质可以说是 CoffeeScript 的一个巨大优势。

基于这些优势，以及我们后面要介绍的 CoffeeScript 所具有的其他优秀性质，Ruby on Rails 从 3.1 版本开始，正式采纳了 CoffeeScript。

① Processing 是一种用于视觉设计和绘图的开源编程语言（及 IDE 开发环境），发布于 2001 年。

安装方法

CoffeeScript 的安装方法有好几种，在 Ubuntu 等 Debian 系 Linux 环境中，可以像平常一样作为软件包进行安装。

```
$ sudo apt-get install coffeescript ↵
```

↵表示换行

或者，可以使用 node.js（参见 6.4 节），通过它的软件包系统 npm 来进行安装。

```
$ sudo npm install coffee-script ↵
```

除此之外的情况，则可以从 http://coffeescript.org/ 下载 tar.gz 文件。

安装完毕之后就可以使用 coffee 命令了。输入 coffee -h 可以显示命令行选项一览。

基本的用法：

```
$ coffee 程序.coffee ↵
```

（CoffeeScript 程序一般用 .coffee 作为扩展名)可以直接运行文件中保存的 CoffeeScript 程序。要将 CoffeeScript 程序编译为 JavaScript，可以使用“-c”选项。

```
$ coffee -c 程序.coffee ↵
```

结果就会输出一个扩展名替换为 .js 的文件，即编译后的 JavaScript 程序。

声明和作用域

我自己几乎没有用 JavaScript 编程的经验，不过听身为 JavaScript 程序员的好友说，对 JavaScript 的不满之一，就是它的变量声明和作用域。

在 JavaScript 中，局部变量需要用保留字“var”进行显式声明，如果不小心忘记声明，这个变量就会变成全局变量。由于全局变量在任何地方都可以访问和修改，于是就变成一个孕育 bug 的可怕温床。

在 CoffeeScript 中，对这一点进行了反省，对于变量引用的规则做出了一些修改。首先，变量的声明不需要用 var，而是通过赋值来进行。在函数中第一个赋值语句被视为对局部变量的声明，这一点与 Ruby 和 Python 十分相似。例如：

```
  foo = 42
```

在 CoffeeScript 中只是一个单纯的赋值语句，但编译为 JavaScript 后，则变成了：

```
var foo;
foo = 42;
```

CoffeeScript 减少了声明，看上去更加简洁。

由于 CoffeeScript 中通过赋值语句会将所有的变量都声明为局部变量，因此要创建全局变量是不可能的。和 JavaScript 不同，CoffeeScript 中位于顶层的变量不是全局变量，而是局部变量（除非用“-b”选项进行显式指定）。

此外，由于不存在对局部变量的显式声明，因此当外侧作用域中存在同名变量时，则以外侧变量优先。如果无意中使用了同名变量，则有可能产生难以发现的 bug。Ruby 中也有同样的问题，但在 Ruby 1.9 之后版本中，可以通过对代码块作用域固有的局部变量进行显式声明来回避这一问题。

CoffeeScript 中可以在变量名前面加上 @ 来进行引用，这相当于：

```
this.变量名
```

的缩写。对实例变量的引用使用“@”这一点和 Ruby 很像呢。

此外，变量名等末尾还可能出现“?”。这种写法乍一看好像也是从 Ruby 来的，但实际上意思完全不同。Ruby 中如果在方法名末尾加上“?”，表示该方法是一个谓词方法（返回真假值的方法）。而在 CoffeeScript 中，变量名后面加上“?”则表示“该变量为 null 和 undefined 以外的值”。

因此，从这个概念进行类推：

```
a ? b
```

表示当 a 为 null 或 undefined 时则为 b，而：

```
a?()
```

表示当 a 为 null 或 undefined 时则为 undefined，否则将 a 作为函数进行调用，而：

```
a?.b
```

则表示“当 a 为 null 或 undefined 时则为 undefined，否则引用 a 中的 b 这一属性”。

例如，将“a?.b”编译为 JavaScript 后如图 1 所示。undefined 的检查方法非常简单，很容易理解。由于有很多方法在出错或遇到异常时会返回 null 和 undefined，如果使用这个功能的话，可以在出错时跳过后面的处理逻辑，从而让程序变得更加简洁。

```
typeof a === "undefined" || a == undefined ? undefined : a.b;
```

图 1 “a?.b”的编译结果

CoffeeScript 也支持多重赋值，如：

```
[a, b] = [1, 2]
```

则表示将 a 赋值为 1，将 b 赋值为 2。和 Ruby 不同的是，不仅是数组，连字典（map）也可以进行展开式的多重赋值，如：

```
{a, b} = {a: 3, b: 4}
```

表示将 a 赋值为 3，将 b 赋值为 4。此外，还可以指定变量名，如：

```
{a:foo, b:bar} = {a: 3, b: 4}
```

表示将 foo 赋值为 3，将 bar 赋值为 4。

多重赋值看似简单，其实编译为 JavaScript 之后会变得相当复杂（图 2）。

```
var _a, _b, a, b, bar, foo;
// [a,b] = [1,2]
_a = [1, 2];
a = _a[0];
b = _a[1];

//{a:foo, b:bar} = {a: 3, b: 4}
_b = {
  a: 3,
  b: 4
};
foo = _b.a;
bar = _b.b;
```

图 2 多重赋值的编译结果

分号和代码块

个人认为，CoffeeScript 最重要的改善点，就是上面讲到的对声明的省略以及对全局变量问题的解决。然而，看了 CoffeeScript 所编写的程序之后，给我留下最深印象的却并非是上面这一点，而是对分号的省略，以及通过缩进来表现代码块。

在 CoffeeScript 中，像 Python 一样是通过缩进来表现代码块的。例如，匿名函数可以这样写：

```
(a) ->
```

```
console.log(a)
a * a
```

在不必每次都写 function 的同时，还可以将多行的匿名函数用非常简洁的方式表达出来。由于 JavaScript 是将函数作为对象来对待的，因此可以使用高阶函数的编程风格，但匿名函数的表达十分繁琐，经常让人感到非常痛苦。而且，CoffeeScript 中最后一个被求值的表达式会自动成为返回值，和必须写 return 的 JavaScript 相比，程序的表达更加简洁。

值得注意的是，在将包括代码块在内的值作为参数的情形。同样是用缩进来表现代码块的 Python 中，创建匿名函数的 lambda 表达式中，函数体只能采用单一的表达式，而要将多行函数作为对象来使用，则必须先作为局部作用域进行命名和定义，这个规则显得相当麻烦。

作为后起之秀，CoffeeScript 自然考虑到了这个问题，只要用括号整个括起来，就可以当做表达式来使用了。例如，像下面这样：

```
something(((a)->
  console.log(1)
  a * a), 2)
```

缩进表现的代码块不仅可以用于匿名函数，对 if 和 while 结构同样有效。例如：

```
if cond()
  1
else
  2
```

这样的块状结构，当程序体只有一行时就可以在一行中进行表达，如：

```
sq = (a) -> a*a
```

或者是：

```
a = if cond() then 1 else 2
```

正如上述例子中所示，CoffeeScript 中的 if 语句实际上是一个表达式，可以返回值。因此，“~?~:~”这样的三项操作符就没有必要使用，作废了。

省略记法

正如缩进表现的代码块一样，CoffeeScript 的设计方针是让表达尽量简洁。例如，JavaScript 中为了分隔语句而必须使用分号，在 CoffeeScript 中则完全不需要使用分号。

函数调用中包围参数的括号，当只有一个参数时也可以省略。不过，当一个参数都没有的

时候，就无法区分到底是调用函数呢，还是对函数对象进行引用。因此这种情况下，在调用函数时，还是要加上 ()（图 3）。

CoffeeScript 中对象的括号也是可以省略的（图 4）。

```
# 参数的括号可以省略
console.log("hello")
console.log "hello"
a = -> 1
a   # 返回1的函数对象
a() #     调用函数，返回1
```

图 3　函数调用的情况

```
# JavaScript的写法
obj = {a:1, b:2}
# 省略括号
obj = a:1, b:2
# 用换行和缩进来表现
# 逗号也省略了
obj =
  a:1
  b:2
```

图 4　对象的括号可以省略

字符串

CoffeeScript 的字符串也很有讲究。首先，Ruby 中也具备的表达式嵌入功能。如下所示，在字符串中用“#{}”包围起来的表达式，它的值会被嵌入到字符串中。

```
name = "Matz"
console.log "Hello #{name}"
```

此外，还可以像 Python 一样，用三重引号来表示跨行字符串（图 5）。

三重引号在需要将类似 XML 这样的长字符串写入程序中的情况下非常有用。有趣的是，在这里 CoffeeScript 是从 Ruby 和 Python 中平等地借鉴它们的功能呢。

话说，CoffeeScript 的注释也和 Ruby、Python 一样是用“#”开头的（JavaScript 是“//”），从三重引号进行类推，“###”就表示直到下一个“###”为止的多行内容全部为注释。

```
# 换行被忽略，值为ab
console.log "a
b"
# 换行有效，值为
#   a
#   b
console.log """a
b
"""
```

图 5　三重引号表示的字符串

数组和循环

CoffeeScript 中的数组也很有讲究。不过很遗憾，数组没办法像对象一样省略外侧的括号。

```
ary = [
 1
 2
]
```

数组也有省略记法，比如看上去很像 Ruby 的范围表达式：

```
[1..4]
```

这种写法表示“从 1 到 4”，编译成 JavaScript 结果如下：

```
[1,2,3,4]
```

不过，如果范围两端的任一端使用变量的话，编译出来就会变成图 6 这样复杂的结果。

大家对 JavaScript 数组的一个不满，就是针对其内容的循环比较难写（图 7）。于是，在 CoffeeScript 中，for~in~ 循环为数组专用，而对于对象成员的访问，则使用另一种 for~of~ 循环来实现。为了让大家理解它们的区别，我们将图 8 中的 CoffeeScript 程序编译成 JavaScript 的结果显示在图 9 中。

```
# a=4; [1..a]
var a, _i, _results;
a = 4;
(function() {
  _results = [];
  for (var _i = 1;
      1 <= a ? _i <= a : _i >=
a;
      1 <= a ? _i++ : _i--) {
    _results.push(_i);
  }
  return _results;
}).apply(this);
```

图 6　数组范围表达式编译结果

```
// (a) 本来是想获取数组的内容
var ary, i;
ary = [7,8,9,0];
for (i in ary) {
  console.log(i);
}
// 结果显示的不是内容而是索引

// (b) 要获取数组的内容应使用如下方法
var _i;
for (_i = 0, _len = a.length; _i < _len; _
i++) {
  i = a[_i];
  console.log(i);
}
// 这样才能真正显示数组的内容

// (c) for~in~原本是面向对象的
var obj;
obj = {foo: 1, bar: 2};
for (i in obj) {
  console.log(i);
}
// 可以取得对象的成员名称
```

图 7　JavaScript 的数组循环

```
ary = [7,8,9,0];
obj = {foo: 1, bar: 2};
# 数组用循环(显示其元素)
for i in ary
  console.log i

# 对象是无法循环的
# 因为它不是数组
for i in obj
  console.log i

# for~of相当于JavaScript的for~in~
# 显示成员名称
for i of obj
  console.log i

# 显示索引
for i of ary
  console.log i
```

图 8　CoffeeScript 的 for 循环

```
var ary, i, obj, _i, _j, _len,
    _len2;
ary = [7, 8, 9, 0];
obj = {
  foo: 1,
  bar: 2
};
for (_i = 0, _len = ary.length;
     _i < _len; _i++) {
  i = ary[_i];
  console.log(i);
}
for (_j = 0, _len2 = obj.length;
     _j < _len2; _j++) {
  i = obj[_j];
  console.log(i);
}
for (i in obj) {
  console.log(i);
}
for (i in ary) {
  console.log(i);
}
```

图 9　图 8 程序的编译结果

类

JavaScript 是基于原型的面向对象语言，因此并不像基于类的语言一样，具备直接支持类定义和方法定义等功能的语法。另一方面，JavaScript 虽然提供了用于从原型生成新对象的 new 语句，但不知为何，作为原型的却是函数对象，总是让人感觉怪怪的。

虽然这也可以说是一种策略吧，不过作为长期以来习惯了基于类的面向对象语言的人来说，多少会觉得痛苦。因此，CoffeeScript 中提供了 class 语句，可以做到看上去像是基于类的面向对象语言。实际上，新版本的 JavaScript 中也提供了 class 语句，但出于兼容性上的考虑，CoffeeScript 并没有直接使用 JavaScript 的 class 语句。

CoffeeScript 的 class 定义如图 10 所示。和 CoffeeScript 的简洁相比，编译为 JavaScript 之后

```
class Person
  # 构造方法
  # Ruby的initialize，Python的__init__
  constructor: (name) ->
    @name = name

# 继承
class SalaryMan extends Person
  constructor: (name, @salary) ->
    # 调用超类的方法
    super(name)
  earn: => console.log "you earn #{@
salary} YEN a month."
salaryman = new SalaryMan("Matz",
100)
```

图 10　CoffeeScript 的类定义

的结果就显得十分复杂。当然，这是让 JavaScript 硬生生配合 CoffeeScript“面子工程”的结果，也许并不能说是一种公平的比较吧。

图 10 中还有一些很有意思的地方，比如在子类的方法中可以像 Ruby 一样使用 super，以及在方法参数中加上“@”就可以不必通过显式赋值来对实例变量进行初始化。

此外，图 10 中还有一点值得注意。在 SalaryMan 类的 earn 方法定义中，用于函数对象的箭头不是“->”而是“=>”。在 CoffeeScript 中，“=>”被称为胖箭头（fat arrow）。

JavaScript 中，目前是通过 this 来表达上下文的，说实话，this 会在哪一个时间点绑定什么这一点有些难以理解。尤其是在事件回调等情况下，在被调用的函数中，this 到底指向哪里，不实际试验一下的话是想象不出来的。用胖箭头定义的函数对象中，其上下文固定为局部上下文；而作为方法进行定义时，this 永远指向其接收器。这样一来关于 this 的麻烦也就消除了。

还有一点，在图 10 的程序中没有体现，那就是类方法究竟应该如何定义。我们可以利用在 class 实体中 this 绑定为正在被定义的类这一点，使用“@”记法即可。即：

```
class Foo
  @number = 0
  @inc: => @number++
  constructor: ->
    Foo.inc()
    console.log Foo.number
```

在这里，@number 是类对象 Foo 的实例变量，@inc 是类方法。要调用类方法，需要像这样：

```
Foo.inc()
```

显式用类名来进行调用。需要注意的是，类对象的实例变量在创建子类时会被复制，但并不共享。也就是说，即使：

```
class Bar extends Foo
Bar.inc()
```

Foo 的实例变量也不会发生变化。

小结

在这里我们无法涵盖 CoffeeScript 的全部特性，除了上面提到的之外，还有很多十分方便的功能。CoffeeScript 给人的印象是，在发挥 JavaScript 优势的同时，为了消除对 JavaScript 的不满，借鉴了 Ruby 和 Python 等多种语言的功能。虽然它比原本过于简单的 JavaScript 更加复杂一些，

但我感觉它的语言设计中体现了一种绝妙的平衡感。抛开利害关系来说，我甚至觉得有些地方比 Ruby 更加优秀。

CoffeeScript 的编译器是通过 JavaScript 编写的，编译结果也是 JavaScript，因此只要有 JavaScript 引擎，无论在任何环境下都可以工作，更何况，现在 JavaScript 引擎可以说遍地都是。CoffeeScript 利用这个优势，无论在服务器端还是客户端，今后其应用范围都会越来越广，可以说是将来值得期待的语言之一。

3.5 Lua

Lua 是由巴西里约热内卢天主教大学的 Roberto Ierusalimschy 等人开发的一种编程语言。据我所知，诞生于南美洲，并在全世界范围内获得应用的编程语言，Lua 应该是第一个。当然，我不知道除了 Lua 之外还有没有其他语言是来自巴西的。

说句题外话，编程语言及其作者的国籍多种多样（表 1），大家可以看出，并不是所有的编程语言都是诞生于美国的，相反，貌似还是欧洲阵营更加强大一些。尤其是从人口比例来看的话，来自北欧的语言设计者比例相当高，而来自南美的只有 Lua，来自亚洲的则只有 Ruby，真是太寂寞了。

表1 编程语言及开发者的国籍

语　言	开　发　者	国　籍
Fortran	John Bacus	美国
C	Dennis Ritchie	美国
Pascal	Niklaus Wirth	瑞士
Simula	Kristen Nygaard	挪威
C++	Bjarne Stroustrup	丹麦
ML	Robin Milner	英国
Java	James Gosling	加拿大
Smalltalk	Alan Kay	美国
Perl	Larry Wall	美国
Python	Guido van Rossum	荷兰
PHP	Rasmus Lerdof	丹麦
Ruby	松本行弘	日本
Eiffel	Bertrand Meyer	法国
Erlang	Joe Armstrong	瑞典
Lua	Roberto Ierusalimschy	巴西

话说 Lua 这个词，在葡萄牙语中是“月亮”的意思。Lua 的特征是一种便于嵌入应用程序中的通用脚本语言。和它设计思想相似的语言还有 Tcl（Tool Command Language）。Tcl 的语言规格被控制在极小的规模，数据类型也只有字符串型一种，而 Lua 却具备了所有作为通用脚本语言的功能。

从实现上来说，Lua 的解释器是完全在 ANSI C 的范围内编写的，实现了极高的可移植性。另外，Lua 的高速虚拟机实现非常有名，在很多虚拟机系性能评分中都取得了优异的成绩。

示例程序

首先，图 1 是一个简单的 Lua 示例程序，这个程序可以计算阶乘。

```
--在Lua中以“--”开始的行为单行注释
--阶乘计算程序
function fact(n)
 if n == 1 then
  return 1
 else
  return n * fact(n -1)
 end
end

print(fact(6))
```

图 1　计算阶乘的 Lua 程序

怎么样？ end 的用法等等是不是有点 Ruby 的感觉呢？说实话，我在写 Lua 程序的时候，经常会和 Ruby 搞混，从而在一些细节的差异上中招。例如，定义不是 def 而是 function、then 是不能省略的、调用函数时参数周围的括号是不能省略的，等等。

Lua 的语法虽然看起来有点像 Ruby，但其内涵更像 JavaScript。例如对象和散列表是不区分的、对象系统是使用函数对象的等。观察一下 Lua 的行为，就会觉得处处都有 JavaScript 的影子。

数据类型

作为通用脚本语言，Lua 可以操作下列数据类型：

- 数值型
- 字符串型
- 函数型
- 表型
- 用户自定义型

其中数值型和字符串型是在各种语言中都具备的普遍数据类型。值得注意的是，Lua 中的数值型全部是浮点数型，并没有整数型。在其他语言中（如果不特别声明的话），都是采用了和 Perl 相同的设计。函数和表我们稍后会逐一讲解。

用户自定义类型，是指用 C 语言等对 Lua 进行扩展时所使用的数据类型。文件输入输出等功能也是以用户自定义类型的方式来提供的。

函数

在 Lua 中，函数是和字符串、数值、表并列的基本数据结构。Lua 的函数属于第一类对象（first-class object），也就是说和数值等其他类型一样，可以赋值给变量、作为参数传递，以及作为返回值接收。

要获得作为值的函数，需要使用不指定名称的 function 语句。像图 2 中所示的指定名称的 function 语句，和用不指定名称的 function 语句所创建的函数进行赋值之后的结果是一样的。

在 Lua 中，通过最大限度利用其函数对象，就实现了面向对象编程等功能。关于用 Lua 实现面向对象编程的内容，我们稍后会进行讲解。

```
-- 函数a的定义
function a(n)
 print(n)
end

-- 赋值给变量a
a(5)            --> 5
-- 通过type(a)获取a的数据类型
print(type(a)) --> function

-- 通过赋值语句定义函数
b = function (n)
 print(n)
end

-- 可以和a一样进行调用
b(5)            --> 5
-- 检查b的类型
print(type(b)) --> function
```

图 2　函数对象

```
--作为数组的表
array = {1, 2, 3}

 --Lua的数组索引是从1开始的
print(array[1])--> 1
--#是用来求长度的操作符(很像Perl)
print(#array) --> 3

--作为散列表的表
hash = {x = 1, y = 2, z = 3}
 --取出散列表元素
print(hash['x']) --> 1
 --取出散列表元素(结构体风格)
print(hash.y) --> 2

--数组和散列表的共存
 --通过赋值添加成员
array["x"] = 42
 --通过赋值添加成员(结构体风格)
array.y= 55
 --无论哪种方式都可以访问
print(array["x"]) --> 42
print(array.x) --> 42
 --散列表元素长度不包含在“长度”中
print(#array) --> 3
```

图 3　表编程

表

Lua 中最具特色的数据类型是表（table）。表是一种可以实现在其他语言中数组、散列表、对象所有这些功能的万能数据类型。JavaScript 中也是用散列表来实现对象的，但数组又是另外的实现方式，因此 Lua 这种将散列表和数组进行合并的做法显得更加彻底。像这样将数组和散

列表合为一体的语言，除了 Lua 以外还有 PHP。

关于数组和散列表的合并到底是不是一个良好的选择，应该说还有讨论的余地，但至少在减少语言构成要素这一点上是有贡献的。由于 Lua 是动态类型语言，因此无论是变量还是数组的元素都可以存放任意类型的数据。

Lua 中各种表的使用方法如图 3 所示。需要注意的是，有一点和其他很多语言都不太一样，那就是作为数组的表索引是从 1 开始的。的确，以前的 FORTRAN 和 Pascal 中，数组的索引也是从 1 开始的，但从 C 语言之后，最近的语言中，索引基本上都是从 0 开始了，因此很容易搞错。特别是如果不小心添加了一个索引为 0 的元素时，这个元素就不是作为一个数组元素，而是作为一个键为 0 的散列表元素来建立的，这一点很容易中招。

此外，还有一点比较违背直觉，那就是在 Lua 中对表应用获取长度的操作符“#”时，其“长度”只包括以（正）整数为索引的数组元素，而表中的散列表元素则不包含长度中，这一点是需要注意的。

元表

Lua 本来不是设计为一种面向对象的语言，因此其面向对象功能是通过元表（meta table）这样一种非常怪异的方式来实现的。Lua 中并不直接支持面向对象语言中常见的类、对象和方法，其中对象和类是通过表来实现，而方法是通过函数来实现的。

首先我们来讲讲元表到底是什么。在 Lua 中，表和用户自定义数据类型中有元表这样一种设定，这个设定通过 setmetatable() 函数来执行。对于表和用户自定义数据类型来说，在需要执行某项操作时，就会产生与该操作相对应的事件。针对各个事件所进行的处理是根据事件类型而决定的，但对于设定了元表的表和用户自定义类型来说，Lua 会参照元表来进行处理。

例如，在执行表元素引用的 index 事件中，处理逻辑如下。当执行 table[key] 表达式时，首先会确认 table 是实际的表，还是用户自定义数据等其他类型。

如果 table 为实际的表，则不通过元表直接取出与 key 相对应的元素（使用 rawget 函数）。如果表中存在该元素，则该元素即为该事件的执行结果。

如果 key 所对应的元素不存在，则取出 table 的元表，并从 table 的元表中取出 __index 这个元素。如果 __index 元素没有被创建，则返回 nil 结果，结束事件处理。

当 table 不是实际的表时，也会从 table 的元表中取出 __index 元素。如果 __index 元素没有

被创建，则表示 table 不支持 index 事件，产生错误。

如果 __index 元素的值为一个函数，则将 table 和 index 作为参数调用该函数。否则，则忽略 table，直接将该元素（这里假定它为 h）作为表来使用，并从中检索 key 所对应的元素。

如果将上述逻辑用 Lua 编写出来，就是图 4 这样。基本上，对于任何事件，其处理都可以归结为下面的逻辑：

- 如果存在规定的操作则执行它。
- 否则，从元表中取出各事件所对应的"__"开头的元素，如果该元素为函数，则调用该函数。
- 如果该元素不为函数，则用该元素代替 table 来执行事件所对应的处理逻辑。

```
--table[key]对于table[key]的事件处理
function gettable_event(table, key)
 local h
 if type(table) == "table" then
  local v = rawget(table, key)
  if v ~= nil then return v end
  h = metatable(table).__index
  if h == nil then return nil end
 else
  h = metatable(table).__index
  if h == nil then
   error(...)
  end
 end
 if type(h) == "function" then
  return (h(table, key)) --call the
handler
 else
  return h[key] --or repeat operation
on it
 end
end
```

图 4　index 事件处理

Lua 所处理的事件一览如表 2 所示。

表2　Lua的事件

事件名	说明	备注
add	加法（+）	以从左到右的顺序搜索元表
sub	减法（–）	同上
mul	乘法（*）	同上
div	除法（/）	同上
mod	求余（%）	同上
pow	幂（^）	同上
unm	单项减法（–）	—
concat	连接（..）	以从左到右的顺序搜索元表
len	求长度（#）	—
eq	比较（==）	数值、字符串则直接比较
lt	小于（<）	大于（>）则两侧对调
le	小于等于（<=）	如果没有 __le 则搜索 __lt
index	引用元素（[]）	x["key"]和 x.key 都会触发该事件
newindex	设定元素（[]=）	同上
call	函数调用	—

方法调用的实现

说了这么多，元表到底该如何使用，到底能实现哪些面向对象编程，好像还是不明白呢。

面向对象编程的基本就是创建对象和调用方法。Lua 中，原则上表是作为对象来使用的，因此创建对象没有任何问题。关于调用方法，如果表的元素为函数对象的话，则可以直接调用。

Lua 中可以这样写：

```
obj.x(foo)
```

这表示从 obj 变量所存放的表中取出以 x 为键的值,并将该值视为函数进行调用。这样一来，看上去就和其他面向对象语言中的方法调用一模一样了。

不过，如果将这种方式作为方法调用来考虑的话，从面向对象来说还有几个问题。

首先，obj.x 这种调用方式，说到底只是将表 obj 的属性 x 这个函数对象取出来而已。而在大多数面向对象语言中，方法的实体是位于类中，而不是位于每个单独的对象中。在 JavaScript 等基于原型的语言中，是以原型对象来代替类进行方法的搜索，因此每个单独的对象也并不拥有方法的实体。

因此，在 Lua 中，为了实现这样的方法搜索机制，需要使用元表的 index 事件。像图 4 中说明的一样，只要在元表中设定 __index，当 key 不存在时就会利用 __index 来进行搜索。

于是，只要编写图 5 这样的程序，就可以将对象的方法搜索转发到其他的对象。

```
proto = {
 x = function() print("x") end
}
obj= {}
setmetatable(obj, {__index = proto})
obj.x()
```

图 5　使用元表实现方法搜索

w 在这种情况下，proto 就变成了原型对象，当 obj 中不存在的属性被引用时，就会去搜索 proto。这样一来，类似 JavaScript 这样基于原型的面向对象编程的基础就完成了。

不过，这样还是有问题。通过方法搜索所得到的函数对象只是单纯的函数，而无法获得最初调用方法的表（接收器）相关的信息。于是，过程和数据就发生了分离，无法顺利实现面向对象的功能。

JavaScript 中，这一关于接收器的信息可以通过 this 来访问。此外，在 Python 中通过方法调用的形式所获得的并非单纯的函数对象，而是一个“方法对象”，当调用这个方法对象时，接收器会在内部作为第一参数附加在函数的调用过程中。

那么，Lua 中该怎么办呢？ Lua 准备了一种支持方法调用的语法糖（syntax sugar，指以易读和易写为目的而引入的语法形式）。在 Lua 中，对方法的调用，不推荐使用单纯的元素访问形式，如：

```
obj.x()
```

而是推荐使用这样的形式：

```
obj:x()
```

这就是 Lua 中添加的语法糖，它表示

```
obj.x(obj)
```

的意思。也就是说，通过冒号记法调用的函数，其接收器会被作为第一参数添加进来。不过，表达式 obj 的求值只会进行一次，因此即便对 obj 有副作用，也只会发生一次。冒号记法的函数调用中，如果还有其他参数的话，会相应顺次向后移动一个位置。

也就是说，在 Python 中通过使用方法对象来实现的、将接收器添加为第一参数的函数调用，在 Lua 中是通过采用一种特殊的调用形式来实现的。这样的设计不包含复杂的机制，能够让人感受到 Lua 的简单哲学。

这个语法糖对方法定义也有效，在图 5 的程序中添加下面几行：

```
function base:y(x)
  print(self,x)
end
```

语法糖会解释为下面的代码：

```
base.y = function(self,x)
  print(self)
end
```

从而在 base 中定义了一个方法。

Lua 中进行方法调用特别需要注意的一点，就是用冒号记法定义的方法。在调用的时候也必须使用冒号记法来进行调用，否则，和 self 相当的参数就不会被添加进去，参数的顺序就会错乱，从而无意中成为产生错误的原因。尤其是在其他语言中，方法调用和属性获取大多都采用相同的圆点记法，因此很容易混淆。此外，在 Lua 中，传递给函数的参数个数有差异时并不会出错，因此即便看见错误信息也无法马上发现问题。我也因为在这个问题上中过招而感到很苦恼。

基于原型编程

通过刚才讲解的机制，我们了解了进行面向对象编程所必需的最低限度的功能。然而，说实话，仅有这些功能还不够好用，为了让它更加接近其他的语言，我们再来少许加工一下。

正如刚才讲过的，方法调用通过使用语法糖就可以毫无问题地完成了。而稍显不足的部分，则是对现有对象的扩展机制，也就是说，在像 Ruby 这样基于类的语言中，相当于类和继承的功能。

不过，考虑到 Lua 的对象机制，比起基于类来说，还是基于原型更加合适。于是，我给大家准备了支持基于原型的面向对象编程的一个简单的库（图 6）。这是我自己编写的一个工具，即使不详细了解 Lua 原始的机制，也可以实现面向对象。

```
--Object为所有对象的上级
Object = {}

--创建现有对象副本的方法
function Object:clone()
 --成为新对象的表
 local object = {}
 --复制表元素
 for k,vin pairs(self) do
  object[k] = v
 end

 --设定元表
 --虽然名字叫clone但并不是复制而是向自身转发
 --为了将对类等的修改反映出来
 setmetatable(object, {__index = self)

 return object
end

--以某个对象为原型创建新的对象
--新对象通过initialize方法进行初始化
```

转到右上↗

```
--允许类似基于类编程的用法
function Object:new(...)
 --成为新对象的表
 local object = {}

 --找不到的方法将搜索目标设定为self
 setmetatable(object, {__index = self})

 --和Ruby一样,初始化时调用initialize方法
 --(...)表示将所有参数原样传递
 object:initialize(...)

 return object
end

--实例初始化函数
function Object:initialize(...)
 --默认不进行任何操作
end

--为了本来不需要进行的类似类的操作而准备原型
Class = Object:new()
```

图 6　Lua 面向对象用工具

本来，在基于原型的面向对象编程中，在创建对象时，对于找不到的方法，其转发目标是指定为原型对象的。因此，例如编写一个对图表上的点进行操作的程序的话，只要创建一个具有代表性的点，然后根据需要，通过复制那个点来创建新的点就可以了。

然而，大多数人还是习惯基于类的面向对象编程，因此实际上并不会去创建“具有代表性的点”，而大多会创建一个原型专用的对象，并像类一样去使用它。这个库也考虑到了这一点，因此单独提供了两个方法，一个是从原型创建对象的 new 方法，另一个是创建对象副本的 clone 方法。

clone 方法会返回对象的副本，元表则是参照被复制的对象来进行设定的。这里我们并没有单纯去复制元表，理由是原始对象如果被修改，则需要将修改的部分在副本对象中反映出来。例如，对相当于类的对象中添加了方法的话，我们希望子类中也拥有新添加的方法。

另一方面，new 方法是从原型创建新的对象，并通过 initialize 方法进行初始化。这里调用 initialize 方法的方式，是参考了 Ruby 的设计。

这个工具的使用实例如图 7 所示，每一行程序实际的功能请参见注释。

```
-- 首先创建新的原型
-- 创建表示“点”的原型Point
Point = Class:new()

-- Point实例初始化方法
-- 设定坐标x和y
function Point:initialize(x, y)
  self.x = x
  self.y = y
end

-- 定义Point类的方法magnitude
-- 求与原点之间的距离
function Point:magnitude()
  return math.sqrt(self.x^2 + self.y^2)
end

-- 创建Point类的实例
-- x = 3, y = 4
p = Point:new(3,4)

-- 确认是否设定了成员
print("p.x = ", p.x)  --> p.x = 3
print("p.y = ", p.y)  --> p.x = 4

-- 计算magnitude()
-- 由勾股定理可求得结果为5
print(p:magnitude())  --> 5

-- 继承了Point的Point3D类的定义
-- 为了创建子类要对类进行clone操作
Point3D = Point:clone()

-- Point3D对象的初始化方法
-- 由于变成三维空间因此增加了z轴上的值
```

转到右上↗

```
function Point3D:initialize(x, y, z)
  -- 调用超类的initialize进行初始化
  -- 必须要指定一个self有点不美观
  Point.initialize(self, x, y)
  -- Point3D类的扩展部分
  self.z = z
end

-- Point3D用的magnitude()方法
function Point3D:magnitude()
  return math.sqrt(self.x^2 + self.y^2 +
self.z^2)
end

-- 创建Point3D实例
p3 = Point3D:new(1,2,3)

-- 属性检查
print("p3.x = ", p3.x) --> 1
print("p3.y = ", p3.y) --> 2
print("p3.z = ", p3.z) --> 3

-- 调用magnitude方法
print(p3:magnitude())  --> 3.7416573867739

-- 创建一个p3的副本
p4 = p3:clone()
-- 属性检查
print("p4.x = ", p4.x) --> 1
print("p4.y = ", p4.y) --> 2
print("p4.z = ", p4.z) --> 3

-- 调用magnitude方法的结果相同
print(p4:magnitude())  -->
3.7416573867739
```

图 7　面向对象工具的使用示例

怎么样？这个库既保留了基于原型的风格，又让习惯了基于类的人也能容易使用，我这有点王婆卖瓜自卖自夸的感觉呢。

和 Ruby 的比较（语言篇）

以嵌入为方针进行设计的 Lua，在默认状态下真是简洁得惊人。在标准状态下，除了基本的数据类型之外，其他一概没有。功能上也就是文件输入输出、字符串模板匹配、表操作、几个数学函数，再加上包加载这些而已。和一开始就提供了大量功能的 Ruby 相比，显得有些贫乏。

正如之前所讲过的，Lua 中虽然能够进行面向对象编程，但用元表来进行编程，仿佛感觉是把对象剖开看到五脏六腑一样。这和 Perl 早期的面向对象编程感觉差不多。

话虽如此，我觉得，虽然感觉有些不适，但从实用角度来说，实现面向对象编程还是没有问题的，虽然提供的功能很少，但也并非一无是处。对语言的基本功能上也有一些不满，例如没有异常处理功能等。不过考虑到它是一种应用程序扩展语言，这一点也并非不能妥协。

只能说，功能和大小是需要权衡的吧。不过，Lua 可以通过 C 语言方便地添加功能。默认的功能仅仅是用来表达逻辑，更多的功能只要根据嵌入的应用程序的需求进行添加就可以了。

同样是以嵌入为方针的 Tcl，由于其附带的 GUI 库 Tk 的完成度相当好，而且出乎作者意料的是，更多地不是用于嵌入应用程序，而是直接作为 GUI 语言来使用。不过 Lua 目前还是遵照着最初的目的，以嵌入为主要的应用领域。

嵌入式语言 Lua

Lua 作为嵌入式语言，观察一下它的实现，会发现最具特点的部分是关于解释器的数据都包括在一个名为 lua_State 的结构体中。通过这样的方式，在一个进程中就可以容纳多个解释器，例如为游戏的角色各自分配一个独立的解释器（只要不在意内存用量的话）也是完全可以做到的。

此外，在多线程环境中，通过为每个线程分配解释器，可以最大限度发挥多核的性能。

再有，Lua 的垃圾回收也很有讲究。在游戏之类对实时性要求很高的环境中，对不再使用的对象进行回收的垃圾回收机制一直是个难题。例如，在射击游戏中，如果由于垃圾回收使玩家有 1 秒钟无法操作自己的飞机，那游戏就变得没法玩了。在嵌入环境中，虽然处理性能非常重要，但响应速度更加重要。Lua 通过使用增量垃圾回收的算法，使得程序的暂停时间得到大大缩短。

以轻量、高速、响应快为特色的 Lua，被用于嵌入到各种应用程序中。例如，网络游戏《魔兽世界》（World of Warcraft）、《仙境传说》（Ragnarok Online），以及美国 Adobe Systems 公司开发的图像处理软件“Adobe Photoshop Lightroom”等。也许是出于这个原因，Adobe 对 Lua 的开发提供了赞助。

比较罕见的是，Yamaha 的路由器 RTX1200 中也嵌入了 Lua。以前，美国 Cisco Systems 公司的路由器中曾经嵌入了 Tcl，但 Lua 似乎成为最近的流行趋势了。

除了上述这些以外，嵌入了 Lua 的游戏、Web 服务器、工具软件等应用程序，已然不计其数。

和 Ruby 的比较（实现篇）

如果从嵌入到应用程序这一角度来比较一下 Ruby 和 Lua 的话，则不能否认 Ruby 处在稍许不利的地位。

Ruby 原本是作为独立通用语言，以追求功能和易用性为目标而设计的。将其嵌入到其他应用该程序中，虽说不是不可能，但并非十分擅长，尤其对于缺少用于表现解释器的结构体来说，是一个很大的缺陷。出于这个原因，在一个进程中只能容纳一个解释器。像 Lua 很容易做到的对多个解释器的协调和对线程的分配，在 Ruby 中就非常困难。

有一个叫做 MVM（Multiple VM）的补丁可以解决这个问题，不过要将它引入到标准实现中还需要一些时间。

当然，嵌入了 Ruby 的应用程序也并非一个都没有。例如 Enterbrain 公司发售的软件《RPG 制作大师》[①]中就嵌入了一种叫做 RGSS（RubyGame Scripting System）的 RPG 编程工具，其实体就是 Ruby 1.8。

此外，我还听到过一个比较古老的报告，就是英国的酒吧中放置的“World Cup Trivia”游戏机中，也嵌入了 Ruby。

语言的优劣并不是非黑即白的。在语言本身的功能以及易用性方面 Ruby 更加优秀，我对此有足够的信心。但如果涉及到嵌入到应用程序中，Lua 在实现方面的实力更强，这一点是不能否认的。

① RPG 制作大师（RPG Maker）是由 Enterbrain 公司发售的一款 RPG（角色扮演游戏）制作工具。从 1990 年开始，这个系列已经在不同的平台上推出了数十个版本，RGSS 的诞生和加入则是从 2004 年的“RPG 制作大师 XP”版本开始的。

嵌入式 Ruby

不过，伴随计算机性能的提升，在嵌入式领域中，CPU 性能和内存容量等指标也比以前有了大幅度的改善。控制器等组件的性能已经相当于不久之前的 PC，即便是游戏机，其性能也已经和 PC 不相上下了，而可以作为电脑来使用的手机，其性能、容量也在不断增加。

其结果，就是所谓嵌入式领域中的软件，其复杂性也和以前不可同日而语。如今，软件的开发效率在嵌入式领域中也成为了一个不可回避的问题。因此，为了提高软件开发效率，其中一个有力的工具，就是采用高级的编程语言。

为了这个目的，预计能够在嵌入式领域中大展身手的新 Ruby 引擎开发项目，已经被采纳为日本经济产业省 2010 年度“地域创新研究开发事业”。这个项目的开发由具备丰富嵌入式经验的各界人士共同进行，核心部分的开发工作由我来完成。这个轻型 Ruby 基于 MIT 授权协议进行开源化，通过“mruby”这个名称便可以获得源代码，请大家访问 https://github.com/mruby/mruby。

这个轻型 Ruby 并非用来替代现有的 Ruby，而是以嵌入式领域为中心，对现在的 Ruby（CRuby）所做的补充。就像 JRuby 是面向 JVM 平台对 Ruby 语言可能性所做的扩展一样，新的轻型 Ruby 将是面向嵌入式领域，对 Ruby 可能性所做的又一次扩展。

伴随着嵌入式计算机性能的提升，软件的复杂化也逐步推进，像 Lua 这样以小规模的引擎面向应用程序嵌入的语言，今后的舞台应该会越来越广阔。

“编程语言的新潮流”后记

在本章中，我们着重介绍了一些（在原稿写成的时候）比较新颖的语言。世界上到底有多少种编程语言，具体的数字没人知道。算上个人兴趣制作的，或者以学者撰写论文的一部分而开发的语言的话，恐怕真的是不计其数了吧。实际上，光我上学时所属的研究室，在几年的时间里就开发了三四种新语言。我记得自己的毕业论文也是和编程语言（不是Ruby）的设计、实现相关的。

以这样的背景所诞生的语言，大部分会随着作者的毕业、工作，或者随着兴趣的减弱等各种理由，开发逐渐停滞，然后慢慢消亡。在这样“无数的尸体”中，才出现了凤毛麟角般的异类，它们变得广为人知，并得以长期存在下去。

这样诞生的语言，在应用的过程中，也在不断进化。这几年最具有开创性的，莫过于v8和LuaJIT的出现。

v8是Google Chrome浏览器中搭载的JavaScript引擎，LuaJIT是本章中介绍过的面向嵌入环境的脚本语言Lua的高速实现。这两者都是作为动态语言以拥有惊人的速度为特点，其速度甚至超越了静态类型语言所擅长的编译式引擎的性能。动态编程语言的实现者（包括我在内）一直以来都以“由于没有编译时能利用的类型信息，加上语言的性质是动态的，因此高速实现是很困难（约等于不可能）的”作为借口。而现实中超高速引擎已经出现了，也就再也谈不上什么“不可能”了。今后，以此为基础，大家又要开始新一轮的进步。实际上，各种浏览器的JavaScript性能在这几年间获得了飞跃性的提高，已然进入了一个大竞争的时代。

今后的语言，需要追求兼具动态语言的灵活性和编译式语言的高速性。本章中介绍的Dart，其诞生的背景就在于此。是像Dart这样采用非强制性类型信息，还是像某种静态类型语言一样采用类型推导，这正是语言进化方向中有意思的地方。

我在这几年中也酝酿了关于语言的一些构思。例如像Ruby一样为动态语言，但局部变量无法再次赋值（单一赋值），提供的数据结构基本上不可改变（immutable）等。这些都是函数型语言（尤其是Erlang）的特征，如果和Ruby这样的面向对象功能结合起来的话，我想是不是能形成一种容易调试、又容易发挥多核性能的语言呢？由于我自己忙于开发Ruby，没有精力再着手开发新的语言，各位读者之中有没有人想要挑战一下呢？

第4章 云计算时代的编程

4.1 可扩展性

根据美国加州大学伯克利分校所做的一项名为“How Much Information?”的调查结果，2002年人类新创造的数据总量已超过5艾字节（EB）。其中艾（Exa,艾克萨）是10的18次方，或者说是2的60次方的前缀。这类前缀还有很多,按顺序分别为千(Kilo,10的3次方)、兆(Mega，10的6次方)、吉（Giga，10的9次方）、太（Tera，10的12次方）、拍（Peta，10的15次方）、艾（Exa，10的18次方）。

此外，根据这项调查做出的预测，2006年人类的信息总量可达到161EB，2010年可达到约988EB（约等于1ZB，Z为Zetta，即10的21次方字节）。这意味着，人类在1年内所产生并记录的数据量，已经超过了截止到20世纪末人类所创造的全部信息的总量。

如此大量的信息被创造、流通和记录，这被称为信息爆炸。生活在21世纪的我们，每天都必须要处理如此庞大的信息量。

信息爆炸并不仅仅是社会整体所面临的问题，我们每个人所拥有的数据每天也在不断增加。在我最早接触计算机的20世纪80年代初，存储媒体一般采用5英寸软盘。面对320KB的“大容量”，当时还是初中生的我曾经感叹到：这些数据容量恐怕一辈子都用不完吧。

然而，在20多年以后，我所使用的电脑硬盘容量就已经有160GB之多，相当于5英寸软盘的50万倍。更为恐怖的是，这些容量的8成都已经被各种各样的数据所填满了。刚刚我查了一下，就光我手头保存的电子邮件，压缩之后也足足有3.7GB之多，而这些邮件每天还在不断增加。

信息的尺度感

在物理学的世界中，随着尺度的变化，物体的行为也会发生很大的变化。量子力学所支配的原子等粒子世界中，以及像银河这样的天文学世界中，都会发生一些在我们身边绝对见不到的现象。

在粒子世界中，某个粒子的存在位置无法明确观测到，而只能用概率论来描述。据说，这是因为要观测粒子，必须要通过光（也就是光子这种粒子）等其他粒子的反射才能完成，而正

是这种反射，就干扰了被观测粒子在下一瞬间的位置。

不仅如此，在量子力学的世界中，仿佛可以无视质量守恒定律一样，会发生一些神奇的现象，比如从一无所有的地方产生一个粒子，或者粒子以类似瞬间移动的方式穿过毫无缝隙的墙壁等，这真是超常识的大汇演。

天文世界也是一样。两端相距数亿光年的银河星团，以及由于引力太强连光都无法逃出的黑洞，这些东西仅凭日常的感觉是很难想象的。

这些超乎常理的现象的发生，是因为受到了一些平常我们不太留心的数值的影响。例如光速、原子等粒子的大小、时间的尺度等，它们的影响是无法忽略的。

在 IT 世界中也发生了同样的事情。从小尺度上来说，电路的精密化导致量子力学的影响开始显现，从而影响到摩尔定律；从大尺度上来说，则产生了信息爆炸导致的海量数据问题。

和人不同，计算机不会感到疲劳和厌烦，无论需要多少时间，最终都能够完成任务。然而，如果无法在现实的时间范围内得出结果，那也是毫无用处的。当数据量变得很大时，就会出现以前从来没有考虑过的各种问题，对于这些问题的对策必须要仔细考量。

下面我们以最容易理解的例子，来看一看关于数据保存和查找的问题。

大量数据的查找

所谓查找，就是在数据中找出满足条件的对象。最简单的数据查找算法是线性查找。所谓线性查找其实并不难，只要逐一取出数据并检查其是否满足条件就可以了，把它叫做一种算法好像也确实夸张了一些。

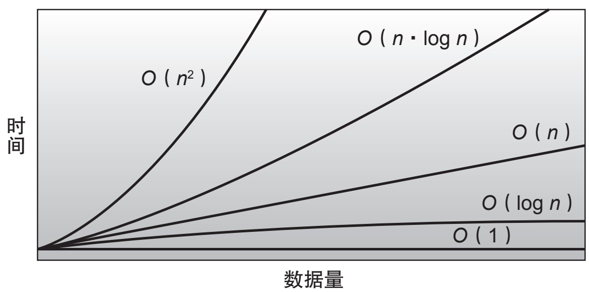


图 1 算法计算性能的差异

线性查找的计算量为 $O(n)$①，也就是说，和查找对象的数据量成正比。在算法的性能中，还有很多属于 $O(n^2)$、$O(n \cdot \log n)$ 等数量级的，相比之下 $O(n)$ 还算是好的（图 1）。

① 这里的记法被称为 O 记法（或者大 O 记法），指的是计算量随参数 n（大多数情况下为输入的数据量）的变化情况。（原书注）

即便如此，随着数据量的增加，查找所需的时间也随之不断延长。假设对 4MB 的数据进行查找只需要 0.5 秒，那么对 4GB（=4000MB[①]）的查找计算就需要 8 分 20 秒，这个时间已经算比较难以忍受的了。而如果是 4TB（=4000GB）的数据，单纯计算的时间就差不多需要 6 天。

像 Google 等搜索引擎所搜索的数据量，早已超过 TB 级，而达到了 PB 级。因此很明显，采用单纯的线性查找是无法实现的。那么，对于这样的信息爆炸，到底应该如何应对呢?

二分法查找

从经验上看，计算性能方面的问题，只能用算法来解决，因为小修小补的变更只能带来百分之几到百分之几十的改善而已。

在这里，我们来介绍一些在一定前提条件下，可以极大地改善查找计算量的算法，借此来学习应对信息爆炸在算法方面的思考方式。

对于没有任何前提条件的查找，线性查找几乎是唯一的算法，但实际上，大多数情况下，数据和查找条件中都存在着一定的前提。利用这些前提条件，有一些算法就可以让计算量大幅度减少。首先，我们来介绍一种基本的查找算法——二分法查找（binary search）。

使用二分法查找的前提条件是，数据之间存在大小关系，且已经按照大小关系排序。利用这一性质，查找的计算量可以下降到 $O(\log n)$。

线性查找大多数是从头开始，而二分法查找则是从正中间开始查找的。首先，将要查找的对象数据和正好位于中点的数据进行比较，其结果有三种可能：两者相等；查找对象较大；查找对象较小。

如果相等则表示已经找到，查找就结束了。否则，就需要继续查找。但由于前提条件是数据已经按照大小顺序进行了排序，因此如果查找对象数据比中点的数据大，则要找的数据一定位于较大的一半中，反之，则一定位于较小的一半中。通过一次比较就可以将查找范围缩小至原来的一半，这种积极缩小查找范围的做法，就是缩减计算量的诀窍。

这个算法用 Ruby 编写出来如图 2 所示。图 2 中定义的方法接受一个已经排序的数组 data，和一个数值 value。如果 value 在 data 中存在的话，则返回其在 data 中的元素位置索引，如果不存在则返回 nil。

---

① 关于 K、M、G 等表示数量级的前缀，有 1000 倍递增的十进制和 1024 倍递增的二进制两种算法，在表示硬盘容量等场合（为了看起来更多）一般都采用十进制，因此这篇文章中也采用了十进制。其实，如果要明确表示 1024 倍递增的二进制方式，则有另外的 Ki、Mi、Gi 等前缀，如“320KiB”，但这种写法一般不太常见。（原书注）

二分法查找的计算量在 n（= 数据个数）较小时差异不大，但随着 n 的增大，其差异也变得越来越大。

表 1 显示了随着数据个数的增加，$\log n$ 的增加趋势。当只有 10 个数据时，n 和 $\log n$ 的差异为 4.3 倍；但当有 100 万个数据时，差异则达到了 72000 倍。

```
def bsearch(data, value)
  lo = 0
  hi = data.size
  while lo < hi
    mid = (lo + hi) / 2  # Note: bug in
Programming Pearl
    if data[mid] < value
      lo = mid + 1;
    else
      hi = mid;
    end
  end
  if lo < data.size && data[lo] == value
    lo  # found
  else
    nil # not found
  end
end
```

图 2　二分法查找程序

说句题外话，出人意料的是，二分法查找的实现并非一帆风顺。例如，1986 年出版的 Jon Bentley 所著的《编程珠玑》[①]（*Programming Pearls*）一书中，就介绍了二分法查找的算法。虽然其示例程序存在 bug，但直到 2006 年，包括作者自己在内，竟然没有任何人注意到。

表1　$O(n)$和$O(\log n)$的计算量变化

n	$\log n$	$n / \log n$（倍）
10	2.302585092994046	4.3429448190325175
100	4.605170185988092	21.71472409516259
1000	6.907755278982137	144.76482730108395
10000	9.210340371976184	1085.7362047581294
100000	11.512925464970229	8685.889638065037
1000000	13.815510557964274	72382.41365054197

这个 bug 就位于图 2 的第 5 行 Note 注释所在的地方。《编程珠玑》中原始的程序是用 C 语言编写的。在 C 这样的语言中，lo 和 hi 之和有可能会超过正整数的最大值，这样的 bug 被称为整数溢出（integer overflow）。

因此，在 C 语言中，这个部分应该写成

```
mid = lo + ((hi - lo) / 2)
```

来防止溢出。在 1986 年的计算机上，索引之和超过整数最大值的情况还非常少见，因此，在很长一段时间内，都没有人注意到这个 bug。

①《编程珠玑》第 2 版中译本由人民邮电出版社于 2008 年出版，黄倩、钱丽艳译。

再说句题外话的题外话，Ruby 中是没有“整数的最大值”这个概念的，非常大的整数会自动转换为多倍长整数。因此，图 2 的 Ruby 程序中就没有这样的 bug。

散列表

从计算量的角度来看，理想的数据结构就是散列表。散列表是表达一个对象到另一个对象的映射的数据结构。Ruby 中有一种名为 Hash 的内建数据结构，它就是散列表。从概念上来看，由于它是一种采用非数值型索引的数组，因此也被称为“联想数组”，但在 Ruby 中（Perl 也是一样）从内部实现上被称为 Hash。而相应地，Smalltalk 和 Python 中相当于 Hash 的数据结构则被称为字典（Dictionary）。

散列表采用了一种巧妙的查找方式，其平均的查找计算量与数据量是无关的。也就是说，用 O 记法来表示的话就是 $O(1)$。无论数据量如何增大，访问其中的数据都只需要一个固定的时间，因此已经算是登峰造极了，从理论上来说。

在散列表中，需要准备一个“散列函数”，用于将各个值计算成为一个称为散列值的整数。散列函数需要满足以下性质：

- 从数据到整数值（0 ~ N−1）的映射
- 足够分散
- 不易冲突

“足够分散”是指，即便数据只有很小的差异，散列函数的计算结果也需要有很大的差异。“不易冲突”是指，不易发生由不同的数据得到相同散列值的情况。

当存在这样一个散列函数时，最简单的散列表，可以通过以散列值为索引的数组来表现（图 3）。

```
hashtable = [nil] * N              ← 根据元素数量创建数组

def hash_set(hashtable, x, y)      ← 数据存放（将散列值作为索引存入）
  hashtable[hash(x)] = y
end

def hash_get(hashtable, x)         ← 数据取出（将散列值作为索引取出）
  hashtable[hash(x)]
end
```

图 3　最简单的散列表

由于散列值的计算和指定索引访问数组元素所需的时间都和数据个数无关，因此可以得出，散列表的访问计算量为 $O(1)$。

不过，世界上没有这么简单的事情，像图 3 这样单纯的散列表根本就不够实用。作为实用的散列表，必须能够应对图 3 的散列表没有考虑到的两个问题，即散列值冲突和数组溢出。

虽然散列函数是数据到散列值的映射，但并不能保证这个映射是一对一的关系，因此不同的数据有可能得到相同的散列值。像这样，不同的数据拥有相同散列值的情况，被称为“冲突”。作为实用的散列表，必须要能够应对散列值的冲突。

在散列表的实现中，应对冲突的方法大体上可以分为链地址法（chaining）和开放地址法（open addressing）两种。链地址法是将拥有相同散列值的元素存放在链表中，因此随着元素个数的增加，散列冲突和查询链表的时间也跟着增加，就造成了性能的损失。

不过，和后面要讲到的开放地址法相比，这种方法的优点是，元素的删除可以用比较简单且高性能的方式来实现，因此 Ruby 的 Hash 就采用了这种链地址法。

另一方面，开放地址法则是在遇到冲突时，再继续寻找一个新的数据存放空间（一般称为槽）。寻找空闲槽最简单的方法，就是按顺序遍历，直到找到空闲槽为止。但一般来说，这样的方法太过简单了，实际上会进行更复杂一些的计算。Python 的字典就是采用了这种开放地址法。

开放地址法中，在访问数据时，如果散列值所代表的位置（槽）中不存在所希望的数据，则要么代表数据不存在，要么代表由于散列冲突而被转存到别的槽中了。于是，可以通过下列算法来寻找目标槽：

(1) 计算数据（key）的散列值
(2) 从散列值找到相应的槽（如果散列值比槽的数量大则取余数）
(3) 如果槽与数据一致，则使用该槽→查找结束
(4) 如果槽为空闲，则散列表中不存在该数据→查找结束
(5) 计算下一个槽的位置
(6) 返回第 3 步进行循环

由于开放地址法在数据存放上使用的是相对普通的数组方式，和链表相比所需的内存空间更少，因此在性能上存在有利的一面。

不过，这种方法也不是尽善尽美的，它也有缺点。首先，相比原本的散列冲突发生率来说，它会让散列冲突发生得更加频繁。因为在开发地址法中，会将有冲突的数据存放到“下一个槽”中，这也就意味着“下一个槽”无法用来存放原本和散列值直接对应的数据了。

当存放数据的数组被逐渐填满时，像这样的槽冲突就会频繁发生。一旦发生槽冲突，就必须通过计算来求得下一个槽的位置，用于槽查找的处理时间就会逐渐增加。因此，在开放地址法的设计中，所使用的数组大小必须是留有一定余量的。

其次，数据的删除比较麻烦。由于开放地址法中，一般的元素和因冲突而不在原位的元素

是混在一起的，因此无法简单地删除某个数据。要删除数据，仅仅将删除对象的数据所在的槽置为空闲是不够的。

这样一来，开放地址法中的连锁就可能被切断，从而导致本来存在的数据无法被找到。因此，要删除数据，必须要将存放该元素的槽设定为一种特殊的状态，即“空闲（允许存放新数据）但不中断对下一个槽的计算”。

随着散列表中存放的数据越来越多，发生冲突的危险性也随之增加。假设真的存在一种理想的散列函数，对于任何数据都能求出完全不同的散列值[①]，那么当元素个数超过散列表中槽的个数时，就不可避免地会产生冲突。尤其是开放地址法中当槽快要被填满时，所引发的冲突问题更加显著。

无论采用哪种方法，一旦发生冲突，就必须沿着某种连锁来寻找数据，因此无法实现 $O(1)$ 的查找效率。

因此，在实用的散列表实现中，当冲突对查找效率产生的不利影响超过某一程度时，就会对表的大小进行修改，从而努力在平均水平上保持 $O(1)$ 的查找效率。例如，在采用链地址法的 Ruby 的 Hash 中，当冲突产生的链表最大长度超过 5 时，就会增加散列表的槽数，并对散列表进行重组。另外，在采用开放地址法的 Python 中，当三分之二的槽被填满时，也会进行重组。

即便在最好的情况下，重组操作所需的计算量也至少和元素的个数相关（即 $O(n)$），不过，只要将重组的频度尽量控制在一个很小的值，就可以将散列表的平均访问消耗水平维持在 $O(1)$。

散列表通过使用散列函数避免了线性查找，从而使得计算量大幅度减少，真是一种巧妙的算法。

布隆过滤器

下面我们来介绍另一种运用了散列函数的有趣的数据结构——布隆过滤器（Bloom filter）。

布隆过滤器是一种可以判断某个数据是否存在的数据结构，或者也可以说是判断集合中是否包含某个成员的数据结构。布隆过滤器的特点如下：

- 判断时间与数据个数无关（$O(1)$）
- 空间效率非常好

① 这样的理想散列函数被称为完美散列函数。如果事先得知数据的取值范围，则构造完美散列函数是可能的。（原书注）

- 无法删除元素
- 偶尔会出错（！）

“偶尔会出错”这一条貌似违背了我们关于数据结构的常识，不过面对大量数据时，我们的目的是缩小查找的范围，因此大多数情况下，少量的误判并不会产生什么问题。

此外，布隆过滤器的误判都是假阳性（false positive），也就是说只会将不属于该集合的元素判断为属于该集合，而不会产生假阴性（false negative）的误判。像布隆过滤器这样“偶尔会出错”的算法，被称为概率算法（probabilistic algorithm）。

布隆过滤器不但拥有极高的时间效率（$O(1)$），还拥有极高的空间效率，理论上说（假设误判率为 1%），平均每个数据只需要 9.6 比特的空间。包括散列表在内，其他表示集合的数据结构中都需要包含原始数据，相比之下，这样的空间效率简直是压倒性的。

布隆过滤器使用 k 个散列函数和 m 比特的比特数组（bit array）。作为比特数组的初始值，所有比特位都被置为 0。向布隆过滤器插入数据时，会对该数据求得 k 个散列值（大于 0 小于 m），并以每个散列值为索引，将对应的比特数组中的比特位全部置为 1。

要判断布隆过滤器中是否包含某个数据，则需求得数据的 k 个散列值，只要其对应的比特位中有任意一个为 0，则可以判断集合中不包含该数据。

即便所有 k 个比特都为 1，也可能是由于散列冲突导致的偶然现象而已，这种情况下就产生了假阳性。假阳性的发生概率与集合中的数据个数 n、散列函数种类数 k，以及比特数组的大小 m 有关。如果 m 相对于 n 太小，就会发生比特数组中所有位都为 1，从而将所有数据都判定为阳性的情况。

此外，当 k 过大时，每个数据所消耗的比特数也随之增加，比特数组填充速度加快，也会引发误判。相反，当 k 过小时，比特数组的填充速度较慢，但又会由于散列冲突的增多而导致误判的增加。

在信息爆炸所引发的大规模数据处理中，像布隆过滤器这样的算法，应该会变得愈发重要。

一台计算机的极限

刚才我们介绍的二分法查找、散列表和布隆过滤器，都是为了控制计算量，从而在现实的时间内完成大量数据处理的算法。

然而，仅仅是实现了这些算法，还不足以应对真正的信息爆炸，因为信息爆炸所产生的数据，

其规模之大是不可能由一台计算机来完成处理的。最近，一般能买到的一台电脑中所搭载的硬盘容量最大也就是几 TB，内存最大也就是 8GB 左右吧。

在摩尔定律的恩泽下，虽然这样的容量已然是今非昔比，但以数 TB 的容量来完成对 PB 级别数据的实时处理，还是完全不够的。

那该怎么办呢？我们需要让多台计算机将数据和计算分割开来进行处理。一台计算机无法处理的数据量，如果由 100 台、1000 台，甚至是 1 万台计算机进行合作，就可以在现实的时间内完成处理。幸运的是，计算机的价格越来越便宜，将它们连接起来的网络带宽也越来越大、越来越便宜。Google 等公司为了提供搜索服务，动用了好几个由数十万台 PC 互相连接起来所构成的数据中心。“云”这个词的诞生，也反映出这种由多台计算机实现的分布式计算，重要性越来越高。

然而，在数万台计算机构成的高度分布式环境中，如何高效进行大量数据保存和处理的技术还没有得到普及。因为在现实中，能够拥有由如此大量的计算机所构成的计算环境的，也只有 Google 等屈指可数的几家大公司而已。

假设真的拥有了大量的计算机，也不能完全解决问题。在安装大量计算机的大规模数据中心中，最少每天都会有几台计算机发生故障。也就是说，各种分布式处理中，都必须考虑到由于计算机故障而导致处理中断的可能性。这是在一台计算机上运行的软件中不太会考虑的一个要素。其结果就是，相比不包含分布式计算的程序开发来说，高度分布式编程得难度要高出许多。

DHT（分布式散列表）

在分布式环境下工作的散列表被称为 DHT（Distributed Hash Table，分布式散列表）。DHT 并非指的是一种特定的算法，而是指将散列表在分布式环境中进行实现的技术的统称。实现 DHT 的算法包括 CAN、Chord、Pastry、Tapestry 等。

DHT 的算法非常复杂，这种复杂性是有原因的。在分布式环境，尤其是 P2P[①]环境中实现散列表，会遇到以下问题：

- 由于机器故障等原因导致节点消失
- 节点的复原、添加
- 伴随上述问题产生的数据位置查找（路由）困难的问题

因此，基本上数据都会以多份副本进行保存。此外，为了应对节点的增减，需要重新计算

① P2P，是 Peer-to-Peer（点对点）的缩写，是一种无中央服务器的对等式通信架构。

数据的位置。

近年来，运用 DHT 技术，在分布式环境下实现非关系型数据库的键 – 值存储（key-value store）数据库受到越来越多的关注。键 – 值存储的例子包括 CouchDB、TokyoTyrant、Kai、Roma 等。

简单来说，这些数据库是通过网络进行访问的 Hash，其数据分别存放在多台计算机中。它们都有各自所针对的数据规模、网络架构和实现语言等方面的特点。

关于分布式环境下的数据存储，除了键 – 值存储以外，还有像 GFS（Google File System）这样的分布式文件系统技术。GFS 是后面要讲到的 MapReduce 的基础。

GFS 并不是开源的，只能在 Google 公司内部使用，但其基本技术已经以论文的形式公开发表，基于论文所提供的信息，也出现了（一般认为）和 GFS 具备同等功能的开源软件“HFS”（Hadoop File System）。

Roma

作为键 – 值存储数据库的一个例子，下面介绍我参与开发的 Roma。Roma（Rakuten On-Memory Architecture）是乐天技术研究所开发的键 – 值存储数据库，是在乐天公司内部为满足灵活的分布式数据存储需求而诞生的。其特点如下：

- 所有数据都存放在内存中的内存式数据库（In-Memory Database，IMDB）
- 采用环状的分布式架构
- 数据冗余化：所有数据除了其本身之外，还各自拥有两个副本
- 运行中可自由增减节点
- 以开源形式发布

Roma 是由多台计算机构成的，这些节点的配置形成了一个虚拟的环状结构（图 4）。这种圆环状的结构让人联想到罗马竞技场，这也正是 Roma 这个名字的由来。

当客户端需要向 Roma 存储一个键 – 值对时，首先根据键的数据求出其散列值。Roma 中的散列值是一个浮点数，在圆环状的结构中，每个节点都划定了各自所负责的散列值范围，客户端根据散列值找出应该存放该数据

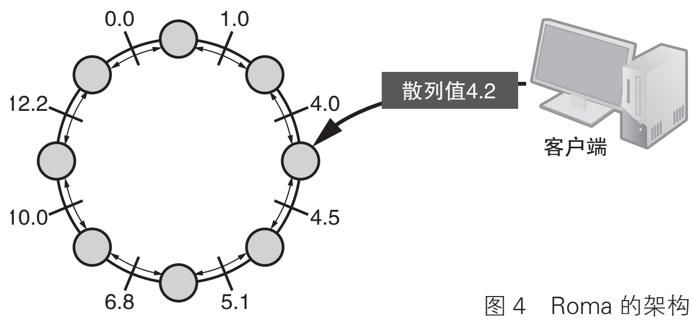


图 4　Roma 的架构

的节点，并向该节点请求存储键所对应的值。由于节点的选择是通过散列函数来计算的，因此计算量是固定的。

Roma 中一定会对数据进行冗余化，所以在数据被写入时，该节点会向其两边相邻的节点发起数据副本请求。因此，对于所有的数据，都会在其负责节点以及两个相邻节点的总共三个节点中保存副本。

Roma 的数据基本上是保存在各个节点的内存中的，但为了避免数据丢失，会在适当的时机以文件的形式输出快照。万一遇到 Roma 系统整体紧急关闭的情况，通过快照和数据写入日志，就可以恢复所有的数据。数据的取出也是同样通过计算散列值找到相应的节点，并对该节点发出请求。

对于像 Roma 这样的分布式键 - 值存储数据库来说，最大的难题在于节点的增减。由大量计算机所构成的系统，必须时刻考虑到发生故障的情况。此外，有时候为了应对数据量和访问量的急剧增加，也会考虑在系统工作状态下增加新节点。

在故障等原因导致节点减少的情况下，一直保持联系的相邻节点会注意到这个变化，并对环状结构进行重组。首先，消失的节点所负责的散列值范围由两端相邻的节点承担。然后，由于节点减少导致有些数据的副本数减少到两个，因此这些数据需要进行搬运，以便保证副本数为三个。

增加节点的处理方法是相同的。节点在圆环的某个地方被插入，并被分配新的散列值负责范围。属于该范围的数据会从两端相邻节点获取副本，新的状态便稳定下来了。

假设，由于网络状况不佳导致某个节点暂时无法访问时，由于数据无法正常复制，可能出现三个数据副本无法保持一致性的问题。实际上，Roma 中的所有数据都通过一种时间戳来记录最后的更新时间。当复制的数据之间发生冲突时，其各自的时间戳必然不同，这时会以时间戳较新的副本为准。

Roma 的优点在于容易维护。只要系统搭建好，节点的添加和删除是非常简单的。根据所需容量增加新的节点也十分方便。

MapReduce

数据存储的问题，通过键 - 值存储和分布式文件系统，在一定程度上可以得到解决，但是在高度分布式环境中进行编程依然十分困难。在分布式散列表中我们也已经接触到了，要解决多个进程的启动、相互同步、并发控制、机器故障应对等分布式环境特有的课题，程序就会变

得非常复杂。在 Google 公司，通过 MapReduce 这一技术，实现了对分布式处理描述的高效化。MapReduce 是将数据的处理通过 Map（数据的映射）、Reduce（映射后数据的化简）的组合来进行描述的。

图 5 是用 MapReduce 统计文档中每个单词出现次数的程序（概念）。实际上要驱动这样的过程还需要相应的中间件，不过这里并没有限定某种特定的中间件。

```
def map（name, document）  ← 接收一个文档
  # name: document name       并分割成单词
  # document: document contents
  for word in document
    EmitIntermediate（word, 1）
  end
end

def reduce(word, values）  ← 对每个单词进行统计
  # key: a word                并返回合计数
  # values: a list of counts of the word
  result = 0
  for v in values
    result += v.to_i
  end
  Emit（result）
end
```

图 5　MapReduce 编写的单词计数程序（概念）

根据图 5 这样的程序，MapReduce 会进行如下处理：

- ❑ 将文档传递给 map 函数
- ❑ 对每个单词进行统计并将结果传递给 reduce 函数

MapReduce 的程序是高度抽象化的，像分配与执行 Map 处理的数据接近的最优节点、对处理中发生的错误等异常情况进行应对等工作，都可以实现高度的自动化。对于错误的应对显得尤其重要，在混入坏数据的情况下，对象数据量如果高达数亿个的话，一个一个去检查就不现实了。

在 MapReduce 中，当发生错误时，会对该数据的处理进行重试，如果依然发生错误的话则自动进行“最佳应对”，比如忽略掉该数据。

和 GFS 一样，MapReduce 也没有开源，但基于 Google 发表的论文等信息，也出现了 Hadoop 这样的开源软件。在 Google 赞助的面向大学生的高度分布式环境编程讲座中，也是使用的 Hadoop。

小结

随着信息爆炸和计算机的日用品化，分布式编程已经与我们近在咫尺，但目前的软件架构可以说还不能完全应对这种格局的变化，软件层面依然需要进化。

4.2 C10K 问题

几年前，我去参加驾照更新的讲座[①]，讲师大叔三令五申“开车不要想当然”。所谓“开车想当然”，就是抱着主观的想当然的心态去开车，比如总认为“那个路口不会有车出来吧”、“那个行人会在车道前面停下来吧”之类的。这就是我们在 2-5 节中讲过的“正常化偏见”的一个例子。作为对策，我们应该提倡这样的开车方式，即提醒自己“那个路口可能会有车出来”、“行人可能会突然窜出来”等。

在编程中也会发生完全相同的状况，比如“这个数据不会超过 16 比特的范围吧”、“这个程序不会用到公元 2000 年以后吧”等。这种想法正是导致 10 年前千年虫问题的根源。人类这种生物，仿佛从诞生之初就抱有对自己有利的主观看法。即便是现在，世界上依然因为“想当然编程”而不断引发各种各样的 bug，包括我自己在内，这真是让人头疼。

何为 C10K 问题

C10K 问题可能也是这种“想当然编程”的副产品。所谓 C10K 问题，就是 Client 10000 Problem，即“在同时连接到服务器的客户端数量超过 10000 个的环境中，即便硬件性能足够，依然无法正常提供服务”这样一个问题。

这个问题的发生，有很多背景，主要的背景如下：

- 由于互联网的普及导致连接客户端数量增加
- keep-alive 等连接保持技术的普及

前者纯粹是因为互联网用户数量的增加，导致热门网站的访问者增加，也就意味着连接数上限的增加。

更大的问题在于后者。在使用套接字（socket）的网络连接中，不能忽视第一次建立连接所需要的开销。在 HTTP 访问中，如果对一个一个的小数据传输请求每次都进行套接字连接，当

① 在日本更新驾照有效期时，必须参加相应的交通讲座，根据驾驶者在上一有效期时间段内有无违反交通法规等情况，讲座分为不同的类型，时间从 30 分钟到 2 小时不等。

访问数增加时，反复连接所需要的开销是相当大的。

为了避免这种浪费，从 HTTP1.1 开始，对同一台服务器产生的多个请求，都通过相同的套接字连接来完成，这就是 keep-alive 技术。

近年来，在网络聊天室等应用中为了提高实时性，出现了一种新的技术，即通过利用 keep-alive 所保持的套接字，由服务器向客户端推送消息，如 Comet，这样的技术往往需要很多的并发连接数。

在 Comet 中，客户端先向服务器发起一个请求，并在收到服务器响应显示页面之后，用 JavaScript 等手段监听该套接字上发送过来的数据。此后，当发生聊天室中有新消息之类的“事件”时，服务器就会对所有客户端一起发送响应数据（图 1）。

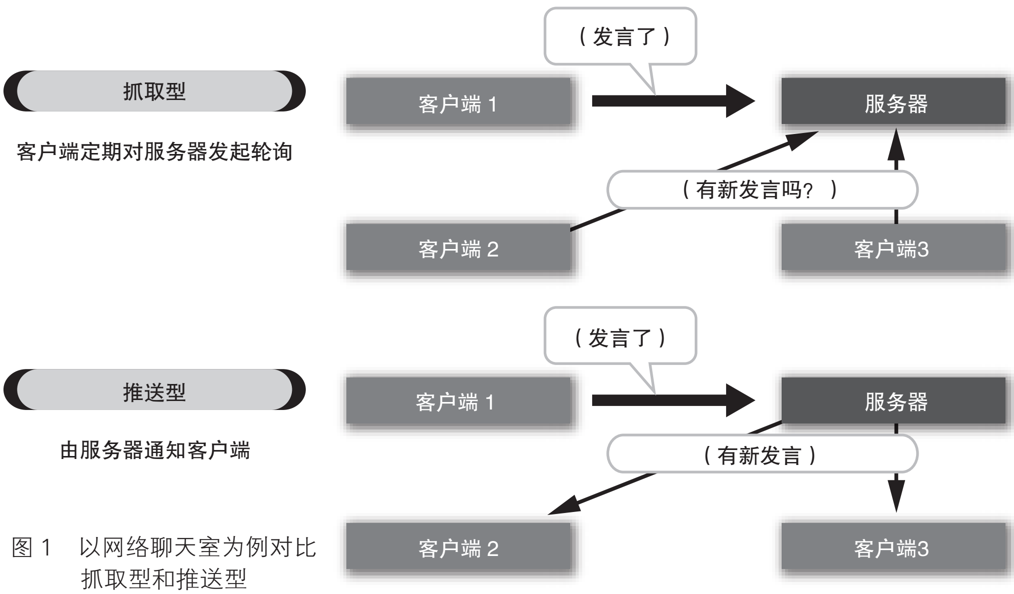


图 1　以网络聊天室为例对比抓取型和推送型

以往的 HTTP 聊天应用都是用抓取型方式来实现的，即以“用户发言”时、“按下刷新按钮”时或者“每隔一定时间”为触发条件，由客户端向服务器进行轮询。这种方式的缺点是，当聊天室中的其他人发言时，不会马上反映到客户端上，因此缺乏实时性。

相对地，Comet 以比较简单的方式实现了高实时性的推送型服务，但是它也有缺点，那就是更多的并发连接对服务器造成的负荷。用 Comet 来提供服务的情况下，会比抓取型方式更早遇到 C10K 问题，从而导致服务缺乏可扩展性。Comet 可以说是以可扩展性为代价来换取实时性的一种做法吧。

C10K 问题所引发的“想当然”

在安全领域有一个“最弱连接”[①]（Weakest link）的说法。如果往两端用力拉一条由很多环（连接）组成的锁链，其中最脆弱的一个连接会先断掉。因此，锁链整体的强度取决于其中最脆弱的一环。安全问题也是一样，整体的强度取决于其中最脆弱的部分。

C10K 问题的情况也很相似。由于一台服务器同时应付超过一万个并发连接的情况，以前几乎从未设想过，因此实际运作起来就会遇到很多“想当然编程”所引发的结果。在构成服务的要素中，哪怕只有一个要素没有考虑到超过一万个客户端的情况，这个要素就会成为“最弱连接”，从而导致问题的发生。

下面我们来看看引发 C10K 问题的元凶——历史上一些著名的“想当然”吧。

> 同时工作的进程数不会有那么多吧。

出于历史原因，UNIX 的进程 ID 是一个带符号的 16 位整数。也就是说，一台计算机上同时存在的进程无法超过 32767 个。实际上，各种服务的运行还需要创建一些后台进程，因此应用程序可以创建的进程数量比这个数字还要小一些。

不过，现在用 16 位整数作为进程 ID 的操作系统越来越少了。比如我手边的 Linux 系统就是用带符号的 32 位整数来作为进程 ID 的。

虽然由数据类型所带来的进程数上限几乎不存在了，不过允许无限地创建进程也会带来很大的危害[②]，因此进程数的上限是可以在内核参数中进行设置的。看一下手边的 Linux 系统，其进程数上限被设定为 48353。

现代操作系统的进程数上限都是在内核参数中设置的，但我们会在后面要讲的内存开销的问题中提到，如果进程数随着并发连接数等比例增加的话，是无法处理大量的并发连接的。这时候就需要像事件驱动模型（event driven model）等软件架构层面的优化了。

而且，Linux 等系统中的进程数上限，实际上也意味着整个系统中运行的线程数的上限，因此为每个并发连接启动一个线程的程序也存在同样的上限。

> 内存的容量足够用来处理所创建的进程和线程的数量吧。

① Weakest link 在中文里一般称为“短板效应”，即一个由很多块木板围城的木桶能装多少水，取决于其中最短的一块木板的长度。

② 如果允许无限制地创建进程的话，那些能够不断产生子进程的程序就会带来持续扩大的危害。（原书注）

进程和线程的创建都需要消耗一定的内存。如果一个程序为每一个连接都分配一个进程或者线程的话，对状态的管理就可以相对简化，程序也会比较易懂，但问题则在于内存的开销。虽然程序的空间等可以通过操作系统的功能进行共享，但变量空间和栈空间是无法共享的，因此这部分内存的开销是无法避免的。此外，每次创建一个线程，作为栈空间，一般也会产生1MB 到 2MB 左右的内存开销。

当然，操作系统都具备虚拟内存功能，即便分配出比计算机中安装的内存（物理内存）容量还要多的空间，也不会立刻造成停止响应。然而，超出物理内存的部分，是要写入访问速度只有 DRAM 千分之一左右的磁盘上的，因此一旦分配的内存超过物理内存的容量，性能就会发生难以置信的明显下滑。

当大量的进程导致内存开销超过物理内存容量时，每次进行进程切换都不得不产生磁盘访问，这样一来，消耗的时间太长导致操作系统整体陷入一种几乎停止响应的状态，这样的情况被称为抖动（thrashing）。

不过，计算机中安装的内存容量也在不断攀升。几年前在服务器中配备 2GB 左右的内存是常见的做法，但现在，一般的服务器中配置 8GB 内存也不算罕见了。随着操作系统 64 位化的快速发展，也许在不久的将来，为每个并发连接都分配一个进程或线程的简单模型，也足够应付一万个客户端了。但到了那个时候，说不定还会产生如 C1000K 问题之类的情况吧。

同时打开的文件描述符的数量不会有那么多吧。

所谓文件描述符（file descriptor），就是用来表示输入输出对象的整数，例如打开的文件以及网络通信用的套接字等。文件描述符的数量也是有限制的，在 Linux 中默认状态下，一个进程所能打开的文件描述符的最大数量是 1024 个。

如果程序的结构需要在一个进程中对很多文件描述符进行操作，就要考虑到系统对于文件描述符数量的限制。根据需要，必须将设置改为比默认的 1024 更大的值。

在 UNIX 系操作系统中，单个进程的限制可以通过 setrlimit 系统调用进行设置。系统全局上限也可以设置，但设置的方法因操作系统而异。

或者我们也可以考虑用这样一种方式，将每 1000 个并发连接分配给一个进程，这样一来一万个连接只要 10 个进程就够了，即便使用默认设置，也不会到达文件描述符的上限的。

要对多个文件描述符进行监视，用 select 系统调用就足够了吧。

正如上面所说的，“一个连接对应一个进程 / 线程”这样的程序虽然很简单，但在内存开销等方面存在问题，于是我们需要在一个进程中不使用单独的线程来处理多个连接。在这种情况下，

如果不做任何检查就直接对套接字进行读取的话，在等待套接字收到数据的过程中，程序整体的运行就会被中断。

用单线程来处理多个连接的时候，像这种等待输入时的运行中断（被称为阻塞）是致命的。为了避免阻塞，在读取数据前必须先检查文件描述符中的输入是否已经到达并可用。

于是，在 UNIX 系操作系统中，对多个文件描述符，可以使用一个叫做 select 的系统调用来监视它们是否处于可供读写的状态。select 系统调用将要监视的文件描述符存入一个 fd_set 结构体，并设定一个超时时间，对它们的状态进行监视。当指定的文件描述符变为可读、可写、发生异常等状态，或者经过指定的超时时间时，该调用就会返回。之后，通过检查 fd_set，就可以得知在指定的文件描述符之中，发生了怎样的状态变化（图 2）。

```
#define NSOCKS 2
int sock[NSOCKS], maxfd;  ←—— sock[1]、sock[2]……中存入要监视的socket。
fd_set readfds;                maxfd中存入最大的文件描述符
struct timeval tv;
int i, n;

FD_ZERO(&readfds);  ←—— fd_set初始化
for (i=0; i<NSOCKS; i++) {
  FD_SET(sock[i], &readfds);
}

tv.tv_sec = 2;  ←—— 2秒超时
tv.tv_usec = 0;

n = select(maxfd+1, &readfds, NULL, NULL, &tv);  ←—— 调用select,这次只监视read。
                                                     关于返回值n:负数-出错，0-超时，
if (n < 0) {  /* 出错 */                             正数-状态发生变化的fd数量
  perror(NULL);
  exit(0);
}
if (n == 0) { /* 超时 */
  puts("timeout");
  exit(0);
}
else {        /* 成功 */
  for (i=0; i<NSOCKS; i++) {
    if (FD_ISSET(sock[i], &fds)) {
      do_something(sock[i]);
    }
  }
}
---
```

图 2　select 系统调用的使用示例（节选）

然而，如果考虑到在发生 C10K 问题这样需要处理大量并发连接的情况，使用 select 系统调用就会存在一些问题。首先，select 系统调用能够监视的文件描述符数量是有上限的，这个上限

定义在宏 FD_SETSIZE 中，虽然因操作系统而异，但一般是在 1024 个左右。即便通过 setrlimit 提高了每个进程中的文件描述符上限，也并不意味着 select 系统调用的限制能够得到改善，这一点特别需要注意。

select 系统调用的另一个问题在于，在调用 select 时，作为参数的 fd_set 结构体会被修改。select 系统调用通过 fd_set 结构体接收要监视的文件描述符，为了标记出实际上发生状态变化的文件描述符，会将相应的 fd_set 进行改写。于是，为了通过 fd_set 得知到底哪些文件描述符已经处于可用状态，必须每次都将监视中的文件描述符全部检查一遍。

虽然单独每次的开销都很小，但通过 select 系统调用进行监视的操作非常频繁。当需要监视的文件描述符越来越多时，这种小开销累积起来，也会引发大问题。

为了避免这样的问题，在可能会遇到 C10K 问题的应用程序中尽量不使用 select 系统调用。为此，可以使用 epoll、kqueue 等其他（更好的）用于监视文件描述符的 API，或者可以使用非阻塞 I/O。再或者，也可以不去刻意避免使用 select 系统调用，而是将一个进程所处理的连接数控制在 select 的上限以下。

使用 epoll 功能

很遗憾，如果不通过 select 系统调用来实现对多个文件描述符的监视，那么各种操作系统就没有一个统一的方法。例如 FreeBSD 等系统中有 kqueue，Solariszh 则是 /dev/poll，Linux 中则是用被称为 epoll 的功能。把这些功能全都介绍一遍实在是太难了，我们就来看看 Linux 中提供的 epoll 这个功能吧。

epoll 功能是由 epoll_create、epoll_ctl 和 epoll_wait 这三个系统调用构成的。用法是先通过 epoll_create 系统调用创建监视描述符，该描述符用于代表要监视的文件描述符，然后通过 epoll_ctl 将监视描述符进行注册，再通过 epoll_wait 进行实际的监视。运用 epoll 的程序节选如图 3 所示。和 select 系统调用相比，epoll 的优点如下：

- ❑ 要监视的 fd 数量没有限制
- ❑ 内核会记忆要监视的 fd，无需每次都进行初始化
- ❑ 只返回产生事件的 fd 的信息，因此无需遍历所有的 fd

通过这样的机制，使得无谓的复制和循环操作大幅减少，从而在应付大量连接的情况下，性能能够得到改善。

实际上，和使用 select 系统调用的 Apache 1.x 相比，使用 epoll 和 kqueue 等新的事件监视

API 的 Apache 2.0，仅在这一点上性能就提升了约 20% ～ 30%。

```
int epfd; ◀—— ①首先创建用于epoll的fd，MAX_EVENTS为要监视的fd的最大数量

if ((epfd = epoll_create(MAX_EVENTS)) < 0) { ◀—— epoll用fd创建失败
  exit(1);
}

struct epoll_event event; ◀—— ②将要监视的fd添加到epoll，根据要监视的数量进行循环
int sock;

memset(&event, 0, sizeof(event)); ◀—— 初始化epoll_event结构体
ev.events  = EPOLLIN; ◀—— 对读取进行监视
ev.data.fd = sock;

if (epoll_ctl(epfd, EPOLL_CTL_ADD, sock, &event) < 0) { ◀—— 将socket添加到
                                                          epoll。fd添加失败
  exit(1);
}

int n, i; ◀—— ③通过epoll_wait进行监视
struct epoll_event events[MAX_EVENTS];

while (1) {
  /* epoll_wait的参数
     第一个:epoll用的fd
     第二个:epoll_event结构体数组
     第三个:epoll_event数组的大小
     第四个:timeout时间(毫秒)
     超时时间为负数表示永远等待 */
  n = epoll_wait(epfd, events, MAX_EVENTS, -1);

  if (n < 0) { ◀—— 监视失败
    exit(1);
  }
  for (i = 0; i < n; i++) { ◀—— 对每个fd的处理
    do_something_on_event(events[i])
  }
}
close(epfd); ◀—— 用一般的close来关闭epoll的fd
```

图 3　epoll_create 的 3 段示例程序

使用 libev 框架

即便我们都知道 epoll 和 kqueue 更加先进，但它们都只能在 Linux 或 BSD 等特定平台上才能使用，这一点让人十分苦恼。因为 UNIX 系平台的一个好处，就是稍稍用心一点就可以（比较）容易地写出具备跨平台兼容性的程序。

于是，一些框架便出现了，它们可以将平台相关的部分隐藏起来，实现对文件描述符的监视。在这些框架之中，我们来为大家介绍一下 libev 和 EventMachine。

libev 是一个提供高性能事件循环功能的库，在 Debian 中提供了 libev-dev 包。libev 是通过在 loop 结构体中设定一个回调函数，当发生事件（可读 / 可写，或者经过一定时间）时，回调函数就会被调用。图 4 展示了 libev 大概的用法。由于代码中加了很多注释，因此大家应该不难对 libev 的用法有个大致的理解。

程序基本上就是对用于监视的对象 watcher 进行初始化，然后添加到事件循环结构体（第 66 ~ 72 行），最后调用事件循环（第 75 行）就可以了。接下来，每次发生事件时，就会自动调用 watcher 中设定的回调函数（第 12 ~ 52 行）。在服务器套接字的回调函数（第 27 ~ 42 行）中，会将已接受的来自客户端连接的套接字添加到事件循环中去。

在这个例子中，只涉及了输出输出以及超时事件，实际上 libev 能够监视表 1 所示的所有这些种类的事件。

表1　libev可监视的事件一览

事　件　名	行　　为
ev_io	输入输出
ev_timer	相对时间（n秒后）
ev_periodic	绝对时间
ev_stat	文件属性的变化
ev_signal	信号
ev_child	子进程的变化
ev_idle	闲置

```
1       /* 首先包含<ev.h> */
2       #include <ev.h>
3
4       /* 其他头文件 */
5       #include <stdio.h>
6       #include <sys/types.h>
7       #include <sys/socket.h>
8
9       ev_io srvsock_watcher;
10      ev_timer timeout_watcher;
11
12      /* 读取socket的回调函数 */
13      static void
14      sock_cb(struct ev_loop *loop, ev_io
        *w, int revents)
15      {
16        /* 读取socket处理 */
17        /* 省略do_socket_read的实现部分 */
18        /* 到达EOF则返回EOF */
19        if (do_socket_read(w->fd) == EOF) {
20          ev_io_stop(w);     /* 停止监视 */
21          close(w->fd);      /* 关闭fd */
22          free(w);           /* 释放ev_io */
23        }
24      }
25
26      /* 服务器socket的回调函数 */
27      static void
28      sv_cb(struct ev_loop *loop, ev_io *w,
        int re
        vents)
29      {
30        struct sockaddr_storage buf;
31        ev_io *sock_watcher;
32        int s;
33
34        /* 接受客户端socket连接 */
35        s = accept(w->fd, &buf,
        sizeof(buf));
36        if (s < 0) return;
37
38        /* 开始监视客户端socket */
39        sock_watcher = malloc(sizeof(ev_
        io));
40        ev_io_init(sock_watcher, sock_cb, s,
        EV_READ);
```

图 4　libev 的用法

libev 可以根据不同的平台自动使用 epoll、kqueue、/dev/poll 等事件监视 API。如果这些 API 都不可用，则会使用 select 系统调用。使用了 libev，就可以在监视并发连接时无需担心移植性了。

在使用像 libev 这样的事件驱动库时，必须要注意回调函数不能发生阻塞。由于事件循环是在单线程下工作的，因此在回调函数执行过程中，是无法对别的事件进行处理的。不仅是 libev，在所有事件驱动架构的程序中，都必须尽快结束回调的处理。如果一项工作需要花费很多时间，则可以将其转发给其他进程 / 线程来完成。

```
    ev_io_start(loop, sock_watcher);
  }

  /* 超时的回调函数 */
  /* 事件循环60秒后调用 */
  static void
  timeout_cb(struct ev_loop *loop, ev_timer *w, int revents)
  {
    puts("timeout");
    /* 结束所有的事件循环 */
    ev_unloop(loop, EVUNLOOP_ALL);
  }

  int
  main(void)
  {
    /* 获取事件循环结构体 */
    /* 一般用default就可以了 */
    struct ev_loop *loop = ev_default_loop(0);

    /* 服务器socket的获取处理 */
    /* 篇幅所限,省略get_server_socket的实现部分 */
    /* socket, bind, 执行socket、bind、listen等*/
    int s = get_server_socket();

    /* 开始监视服务器socket */
    ev_io_init(&srvsock_watcher, sv_cb, s, EV_READ);
    ev_io_start(loop, &srvsock_watcher);

    /* 设置超时时间(60秒) */
    ev_timer_init(&timeout_watcher, timeout_cb, 60.0, 0.0);
    ev_timer_start(loop, &timeout_watcher);

    /* 事件循环体 */
    ev_loop(loop, 0);

    /* unloop被调用时结束事件循环 */
    return 0;
  }
```

图 4　libev 的用法（续）

使用 EventMachine

刚刚我们做了很多底层编程的工作，例题也是用 C 语言来写的。不过，仔细想想的话，正如一开始所讲过的那样，C10K 问题的本质其实是“明明硬件性能足够，但因来自客户端的并发连接过多导致处理产生破绽”。既然我们完全可以不那么在意 CPU 的性能，那是不是用 Ruby 也能够应对 C10K 问题呢？

答案是肯定的。实际上，用 Ruby 开发能应付大量并发连接的程序并不难，支持这一功能的框架也已经有了。下面我们来介绍一种 Ruby 中应对 C10K 问题的事件驱动框架——EventMachine。用 Ruby 的软件包管理系统 RubyGems 就可以很轻松地完成 EventMachine 的安装：

```
$ sudo gem install eventmachine ⏎
```

⏎表示换行

我们用 EventMachine 来实现了一个 Echo（回声）服务器，它的功能就是将写入 socket 的数据原原本本返回来，程序如图 5 所示。图 4 中运用 libev 编写的程序足足有 80 行，这还是在省略了本质的处理部分的情况下，而图 5 的程序完整版也只需要 20 行。由此大家也可以感受到 Ruby 颇高的表达能力了吧。

在 EventMachine 中，回调是以 Ruby 模块的形式进行定义的。在图 5 的例子中，EchoServer 模块扮演了这个角色。这个模块中重写了几个方法，从而实现了回调，也就是一种 Template Method 设计模式吧。实现回调的方法如表 2 所示。

```
require 'eventmachine'

module EchoServer
  def post_init
    puts "-- someone connected to the echo
server!"
  end

  def receive_data data
    send_data data
    close_connection if data =~ /quit/i
  end

  def unbind
    puts "-- someone disconnected from the echo
server!"
  end
end

EventMachine::run {
  EventMachine::start_server "127.0.0.1", 8081,
EchoServer
  }
```

图 5　运用 EventMachine 编写的 Echo 服务器

表2　EventMachine的回调方法

方　法　名	调用条件	目　　的
post_init	socket 连接后	初始化连接
receive_data(data)	数据接收后	读取数据
unbind	连接终止后	终止处理
connection_completed	连接完成时	初始化客户端连接
ssl_handshake_completed	SSL 连接时	SSL
ssl_verify_peer	SSL 连接时	SSL 节点验证
proxy_target_unbound	proxy 关闭时	转发目标

同样是事件驱动框架，但 libev 和 EventMachine 在功能上却有很大的不同。

libev 的目的只是提供最基本的事件监视功能，而在套接字连接、内存管理等方面还需要用户自己来操心。同时，它能够支持定时器、信号、子进程状态变化等各种事件。libev 是用于 C 语言的库，虽然程序可能会变得很繁琐，但却拥有可以应付各种状况的灵活性。

另一方面，EventMachine 提供了多并发网络连接处理方面的丰富功能。从图 5 的程序中应该也可以看出，由于它对套接字连接、数据读取都提供了相应的支持，因此在网络编程方面可

以节约大量的代码，但相对来说，它所支持的事件种类只有输入输出和定时器。

作为 C 语言的库，libev 的功能专注于对事件的监视；而作为面向 Ruby 的框架，EventMachine 则支持包括服务器、客户端的网络连接和输入输出，甚至是 SSL 加密。这也许也反映了两种编程语言性格之间的差异吧。

其实，关于 libev 和 EventMachine 是否真的能够处理大量并发连接，最好是做个性能测试，但以我手上简陋的环境来说，恐怕无法尝试一万个客户端的连接，也不可能为了这个实验准备一万台笔记本电脑吧。而且，要进行可扩展性的实验，还是需要准备一个专门的网络环境才行。不过话说回来，libev 和 EventMachine 都已经在各种服务中拥有一些应用实例，应该不会存在非常极端的性能上的问题吧[①]。

小结

在 libev、EventMachine 等事件驱动框架中，如何尽量缩短回调的执行时间是非常重要的，因为在回调中如果发生输入输出等待（阻塞），在大量并发连接的情况下是致命的。于是，在输入输出上如何避免阻塞（非阻塞 I/O）就显得愈发重要。

① 也有人对 EventMachine 的性能和功能表示不满，于是又开发出了一个新的异步 I/O 库——Celluloid.io。（原书注）

4.3 HashFold

在 4-1 中，我们介绍了大量信息被创造和记录所引发的“信息爆炸”，以及为应付信息爆炸而将处理分布到多台计算机中进行的方法。对于运用多台计算机构成的高度分布式处理环境中的编程模型，我们介绍了美国 Google 公司提出的 MapReduce。下面我们要介绍的 HashFold 正是它的一种变体，Steve Krenzel 在其网站中（http://stevekrenzel.com/improving-mapreduce-with-hashfold）也对此做了介绍。

MapReduce 通过分解、提取数据流的 Map 函数和化简、计算数据的 Reduce 函数，对处理进行分割，从而实现了对大量数据的高效分布式处理。

相对地，HashFold 的功能是以散列表的方式接收 Map 后的数据，然后通过 Fold 过程来实现对散列表元素的去重复。这种模型将 MapReduce 中一些没有细化的部分，如 Map 后的数据如何排序再进行 Reduce 等，通过散列表这一数据结构的性质做了清晰的描述，因此我个人很喜欢 HashFold。不过，虽然我对 HashFold 表示支持，但恐怕它要成为主流还是很困难的。

即便无法成为主流，对于大规模数据处理中分布式处理的实现，HashFold 简洁的结构应该也可以成为一个不错的实例。

HashFold 的 Map 过程在接收原始数据之后，将数据生成 key、value 对。然后，Fold 过程接收两个 value，并返回一个新的 value。图 1 所示的就是一个运用 HashFold 的单词计数程序。

```
def map(document)  ←—— 接收文档并分解为单词
  # document: document contents
  for word in document
    # key=单词,计数=1
    yield word, 1
  end
end

def fold(count1, count2)  ←—— 对单词进行统计
  # count1, count2: two counts for a word
  return count1 + count2
end
```

图 1　用 HashFold 编写的单词计数程序（概念）

单词计数是 MapReduce 主要的应用实例，这个说法已经是老生常谈了，每次提到 MapReduce 的话题，就会把它当成例题来用。而 HashFold 则是单词计数的最佳计算模型，当然，它也可以用来进行其他的计算。

下面我们按照图 1 的概念，来实现一个简单的 HashFold 库。因为如果不实际实现一下的话，我们就无法判断这种模型是否具有广泛的适应性。

为了进行设计，我们需要思考满足 HashFold 性质的条件，于是便得出了以下结论。

首先，由于在 Ruby 中无法通过网络发送过程，因此 HashFold 的主体不应是函数（过程），而应该是对象。如果是对象的话，只要通过某些手段事先共享类定义，我们就可以用 Ruby 中内建的 Marshal 功能通过网络来传输对象了。

我们希望这个对象最好是 HashFold 类的子类。这样一来，HashFold 类所拥有的功能就可以被继承下来，从而可以使用 Template Method 模式来提供每个单独的 Map 和 Fold（图 2）

唔，貌似挺好用的。下面就我们来制作一个满足上述设计的 HashFold 库。

```
class WordCount < HashFold
  def map(document)        ← 分割单词
    for word in document
      yield word, 1
    end
  end

  def fold(count1, count2)     ← 重复的单词进行合并
                                 计算
    return count1 + count2
  end
end
h = WordCount.new.start(documents)     ← 得到结果Hash(单词=>
                                         单词出现数)
```

图 2　HashFold 库 API（概念）

HashFold 库的实现（Level 1）

好，我们已经完成了 API 的设计，现在我们来实际进行 HashFold 库的实现吧。首先，我们先不考虑分布式环境，而是先从初级版本开始做起。

要实现一个最单纯级别的 HashFold 是很容易的。只要接收输入的数据，并对其执行 Map 过程，如果出现重复则通过 Fold 过程来解决。实际的程序如图 3 所示。

在不考虑分布式环境的情况下，HashFold 的实现其实相当容易，这也反映了 HashFold “易于理解”这一特性。

不过，不实际运行一下，就不知道它是不是真的能用呢。于是我们准备了一个例题程序。

图 4 就是为 HashFold 库准备的例题程序，它可以看成是对图 2 中的概念进行具体化的结果。这是一个依照 MapReduce 的传统方式，对单词进行计数的程序。今后我们会对 HashFold 库进

行升级，但其 API 是不变的，因此单词计数程序也不需要进行改动。

```
class HashFold
  def start(inputs)
    hash = {} ←— 保存结果用的Hash
    inputs.each do |input| ←— 对传递给start的各输入数据调用map方法
      self.map(input) do |k,v|
       if hash.key?(k) ←— 在代码块中传递的key和value如果出现重复则调用fold方法
         hash[k] = self.fold(hash[k], v)
       else
         hash[k] = v ←— 如果尚未存在则存放到Hash中
       end
      end
    end
    hash ←— 返回结果Hash
  end
end
```

图 3　HashFold 库（Level 1）

```
class WordCount < HashFold           不需要进行计数的高频英文单词
  STOP_WORDS = %w(a an and are as be for if in is it of or the to with) ←— 将输入的
                                                                           参数作为
                                                                           文件名

  def map(document)
   open(document) do |f| ←— 对文件各行执行操作
      for line in f ←— 将所有标点符号视为分割符(替换成空格)
        line.gsub!(/[!#"$%&\'()*+,-.\/:;<=>?@\[\\\]^_`{\|}~]/, " ") ←— 将一行的内容
                                                                       分割为单词
        for word in line.split ←— 将字母都统一转换为小写
          word.downcase! ←— 高频单词不计数
          next if STOP_WORDS.include?(word) ←— key=单词,计数=1,传递给代码块
          yield word.strip, 1 ←— 解决重复
        end
      end
    end
  end

  def fold(count1, count2) ←— 对单词计数进行简单累加
    return count1 + count2 ←— 命令行参数用于指定要统计单词的文件。随后按照计数将单词倒序排列
                              (从大到小),并输出排在前20位的单词。
  end
end

WordCount.new.start(ARGV).sort_by{|x|x[1]}.reverse.take(20).each do |k,v|


  print k,": ", v, "\n"
end
```

图 4　单词计数程序

将图 3 的库和图 4 的程序结合起来，就完成了一个最简单的 HashFold 程序。我们暂且将这个程序保存在“hfl.rb”这个文件中。

那么，我们来运行一下看看。首先将 Ruby 代码仓库中的“Ruby trunk”分支下的 ChangeLog（变更履历）作为单词计数的对象。运行结果如图 5 所示。

```
% ruby hf1.rb ChangeLog ⏎
ruby: 11960
rb: 11652
c: 10231
org: 7591
lang: 5494
test: 4224
lib: 3804
ext: 3582
2008: 3172
ditto: 2669
dev: 2382
nobu: 2334
nakada: 2313
nobuyoshi: 2313
2007: 1820
h: 1664
matz: 1659
yukihiro: 1648
matsumoto: 1648
tue: 1639
```

图 5　单词计数的运行结果

从这个结果中，我们可以发现很多有趣的内容。比如，ruby 这个单词出现次数最多，这是理所当然的，而出现次数最多的名字（提交者）是 nobuyoshi nakada（2313 次），远远超出位于第二名的我（1648 次）。原来我已经被超越那么多了呀。

除此之外，我们还能看出提交发生最多的日子是星期二。如果查看一下 20 位之后的结果，就可以看出一周中每天提交次数的排名：最多的是星期二（1639 次），然后依次是星期四（1584 次）、星期一（1503 次）、星期五（1481 次）、星期三（1477 次）、星期六（1234 次）和星期日（1012 次）。果然周末的提交比较少呢，但次数最多的居然是星期二，这个倒是我没有想到的。

不过，光统计一个文件中的单词还不是很有意思，我们来将多个文件作为计数对象吧，比如将 ChangeLog 以及其他位于 Ruby trunk 分支中所有的“.c”文件作为对象。我算了一下，要统计的文件数量为 292 个，大小约 6MB，正好我们也可以来统计一下运行时间（图 6）。这里我们的运行环境是 Ruby 1.8.7，Patch Level 174。

```
% time ruby hf1.rb ChangeLog **/*.c ⏎
rb: 31202
0: 17155
1: 13784
ruby: 13205
（中略）
ruby hf0.rb ChangeLog **/*.c  37.89s user 3.89s system 98% cpu
 42.528 total
```

图 6　以多个文件为对象的运行结果（附带运行时间）

我用的 shell 是“zsh”，它可以通过“**/*.c”来指定当前目录下（包括子目录下）所有的 .c 文件。这个功能非常方便，甚至可以说我就是为了这个功能才用 zsh 的吧。

在命令行最前面加上 time 就可以测出运行时间。time 是 shell 的内部命令，因此每种 shell 输出的格式都不同，大体上总会包含以下 3 种信息。

- user：程序本身所消耗的时间
- system：由于系统调用在操作系统内核中所消耗的时间
- total：从程序启动到结束所消耗的时间。由于系统中还运行着其他进程，因此这个时间要大于 user 与 system 之和

从这样的运算量来看，用时 42 秒还算不赖。不过，6MB 的数据量，即便不进行什么优化，用简单的程序来完成也没有多大问题。

作为参考，我用 Ruby 1.9 也测试了一下，所用的是写稿时最新的 trunk，ruby1.9.2dev（2009-08-08trunk 24443）。

运行结果为 user 18.61 秒、system 0.14 秒、total 18.788 秒，也就是说，和 Ruby 1.8.7 的 42.528 秒相比，速度达到了两倍以上（2.26 倍）。看来 Ruby 1.9 中所搭载的虚拟机“YARV（Yet Another Ruby VM）”的性能不可小觑呢。

从此之后，我们基本上都使用 1.9 版本来进行测试，主要是因为我平常最常用的就是 Ruby 1.9。此外，由于性能测试要跑很多次，如果等待时间能缩短的话可是能大大提高（写稿的）生产效率的。

运用多核的必要性

如果程序运行速度变快，恐怕没人会有意见。相反，无论你编写的程序运行速度有多快，总会有人抱怨说“还不够快啊”。这种情况的出现几乎是必然的，就跟太阳每天都会升起来一样。

问题不仅仅如此。虽然 CPU 的速度根据摩尔定律而变得越来越快，但也马上就要遇到物理定律的极限，CPU 性能的提升不会像之前那样一帆风顺了。这几年来，CPU 时钟频率的提升已经遇到了瓶颈，Intel 公司推出的像 Atom 这样低频率、低能耗的 CPU 的成功，以及让普通电脑也能拥有多个 CPU 的多核处理器的普及，这些都是逐步接近物理极限所带来的影响。

此外，还有信息爆炸的问题摆在我们面前。当要处理的数据量变得非常巨大时，光数据传输所消耗的时间都会变得无法忽略了。在 Google 公司所要处理的 PB 级别数据量下，光是数据的拷贝所花费的时间，就能达到“天”这个数量级。

MapReduce 正是在这一背景下诞生的技术，HashFold 也需要考虑到这方面因素而不断提升性能。

所幸的是，我所用的联想 ThinkPad X61 安装了 Intel Core2 Duo 这个双核 CPU，没有理由不充分利用它。通过使用多个 CPU 进行同时处理，即并发编程，为处理性能的提高提供了新的可能性。

目前的 Ruby 实现所存在的问题

然而，从充分利用多核的角度来看，目前的 Ruby 实现是存在问题的。作为并发编程的工具，我们可以使用线程，但 Ruby 1.8 中的线程功能是在解释器级别上实现的，因此只能同时进行一项处理，并不能充分利用多核的性能。在 Ruby 1.9 中，线程的实现使用了操作系统提供的 pthread 库，本来应该是可以利用多核的，但在 Ruby 1.9 中，为了保护解释器中非线程安全的部分而加上了一个称为 GIL（Giant Intepreter Lock）[①]的锁，由于这个锁的限制，每次还是只能同时执行一个线程，看来在 Ruby 1.9 中要利用多核也是很困难的。

那么，如果要在 Ruby 上利用多核，该怎样做呢？一种方法是采用没有加 GIL 锁的实现。所幸，在 JVM 上工作的 JRuby 就没有这个锁，因此用 JRuby 就可以充分利用多核了。不过，我作为 Ruby 的实现者，在这一点上却非要使用 JRuby 不可，总有点“败给它了”的感觉。

通过进程来实现 HashFold（Level 2）

“如果线程不行的话那就用进程好了。”不过，仔细想想就会发现，利用多个 CPU 的手段，操作系统不是原本就已经提供了吗？那就是进程。如果启动多个进程，操作系统就会自动进行调配，使得各个进程分配到适当的 CPU 资源。

这样的功能不利用起来真是太浪费了。首先，我们先来做一个最简单的进程式实现，即为每个输入项目启动一个进程。

为每个输入启动一个进程的 HashFold 实现如图 7 所示。和线程不同，进程之间是不共享内存的，因此为了返回结果就需要用到进程间通信。在这里，我们使用 UNIX 编程中经典的父子进程通信手段 pipe。

基本处理流程很简单。对各输入启动相应的进程，各个文件的单词计数在子进程中进行。计数结果的 Hash 需要返回给父进程，但和线程不同，父子进程之间无法共享对象，因此需要使

① 这个锁的功能是，在多线程环境中，为了保护非线程安全（没有考虑到多线程同时并行运行）的代码，只允许同时运行一个线程。（原书注）

用 pipe 和 Marshal 将对象进行复制并转发。父进程从子进程接收 Hash 后，将接收到的 Hash 通过 fold 方法进行合并，最终得到单词计数的结果。

说到这里，大家应该明白图 7 程序的大致流程了。而作为编程技巧，希望大家记住关于 fork 和 pipe 的用法，它们在使用进程的程序中几乎是不可或缺的技巧。在 Ruby 中，fork 方法可以附带代码块来进行调用，而代码块可以只在子进程中运行，当运行到代码块末尾时，子进程会自动结束。

```
class HashFold
  def hash_merge(hash,k,v) ◀—— 调用fold,由于要调用多次,因此构建在方法中
    if hash.key?(k)
      hash[k] = self.fold(hash[k], v) ◀—— 如果遇到重复则调用fold方法
    else
      hash[k] = v ◀—— 尚未存在则存放到Hash中
    end
  end

  def start(inputs)
    hash = nil
    inputs.map do |input| ◀—— 对传递给start的每个输入进行循环
      p,c = IO.pipe ◀—— 创建用于父子进程间通信用的pipe
      fork do ◀—— 创建子进程(fork),在子进程中运行代码块
        p.close ◀—— 关闭不使用的pipe
        h = {} ◀—— 保存结果用的Hash
        self.map(input) do |k,v| ◀—— 调用map方法,由于完全复制了父进程的内存空间,因此可以看到
父进程的对象(input)
          hash_merge(h,k,v) ◀—— 存放数据,解决重复
        end
        Marshal.dump(h,c) ◀—— 将结果返回给父进程,这次使用Marshal
      end
      c.close ◀—— 这是父进程,关闭不使用的pipe
      p ◀—— 对父进程一侧的pipe进行map
    end.each do |f| ◀—— 读取来自子进程的结果
      h = Marshal.load(f)
      if hash ◀—— 将结果Hash进行合并
        h.each do |k,v|
          hash_merge(hash, k, v)
        end
      else
        hash = h
      end
    end
    hash
  end
end

#单词计数的部分是共通的
```

图 7　运用进程实现的 HashFold

重要的事情总要反复强调一下，fork 的作用是创建一个当前运行中的进程的副本。由于是副本，因此现在可以引用的类和对象，在子进程中也可以直接引用。但是，也正是由于它只是一个副本，因此如果对对象进行了任何变更，或者创建了新的对象，都无法直接反映到父进程中。这一点，和共享内存空间的线程是不同的。

在不共享内存空间的进程之间进行信息的传递有很多种方法，在具有父子关系的进程中，pipe 恐怕是最好的方法了。

pipe 方法会创建两个分别用来读取和写入的 IO（输入输出）。

```
r,w = IO.pipe
```

在这两个 IO 中，写入到 w 的数据可以从 r 中读取出来。正如刚才所讲过的，由于子进程是父进程的副本，在父进程中创建 pipe，并在子进程中对 pipe 进行写入的话，就可以从父进程中将数据读取出来了。作为好习惯，我们应该将不使用的 IO（在这里指的是父进程中用于写入的，和子进程中用于读取的）关闭掉，避免对资源的浪费。

在这个程序中，从子进程传递结果只需要创建一对 pipe，如果需要双向通信则要创建两对 pipe。

好，我们来运行一下看看。将图 7 的 HashFold 和图 4 的单词计数程序组合起来保存为“hf2.rb”，并运行这个程序。在 1.9 环境下的运行结果为 user 0.66 秒、system 0.08 秒、total 11.494 秒。和非并行版的运行时间 18.788 秒相比，速度是原来的 1.63 倍。考虑到并行处理产生的进程创建开销，以及 Marshal 的通信开销，63% 的改善还算是可以吧。

之所以 user 和 system 时间非常短，是因为实际的单词计数处理几乎都是在子进程中进行的，因此没有被算进去。顺便，在 1.8.7 上的运行时间是 25.528 秒，是 1.9 上的 2.25 倍。

然而，仔细看一看的话，这个程序还是有一些问题的。这个程序中，对每一个输入文件都会启动一个进程，这样会在瞬间内产生大量的进程。这次我们对 292 个文件的单词进行计数，创建了 293 个（文件数量 + 管理进程）进程，而大量的进程则意味着巨大的内存开销。如果要统计的对象文件数量继续增加，就会因为进程数量太多而引发问题。

抖动

当进程数量过多时，就会产生抖动现象。

随着大量进程的产生，会消耗大量的内存空间。在最近的操作系统中，当申请分配的内存数量超过实际的物理内存容量时，会将现在未使用的进程的内存数据暂时存放到磁盘上，从表

面上看，可用内存空间变得更多了。这种技术被称为虚拟内存。

然而，磁盘的访问速度和实际的内存相比要慢上几百万倍。当进程数量太多时，几乎所有的时间都消耗在对磁盘的访问上，实际的处理则陷于停滞，这就是抖动。

其实，用 Ruby 只需要几行代码就可以产生大量的进程，从而故意引发抖动，不过在这里我们还是不介绍具体的代码了。

当然，操作系统方面也考虑到了这一点。为了尽量避免发生抖动，也进行了一些优化。例如写时复制（Copy-on-Write）技术，就是在创建子进程时，对于所有的内存空间并非一开始就创建副本，而是先进行共享，只有当实际发生对数据的改写时才进行复制。通过这一技术，就可以避免对内存空间的浪费。

在 Linux 中还有一个称为 OOM Killer（Out of Memory Killer）的功能。当发生抖动时，会选择适当的进程并将其强制结束，从而对抖动做出应对。当然，操作系统不可能从人类意图的角度来判断哪个进程是重要的，因此 OOM Killer 有时候会错杀掉一些很重要的进程，对于这个功能的评价也是毁誉参半。

运用进程池的 HashFold（Level 3）

大量产生进程所带来的问题我们已经了解了。那么，我们可以不每次都创建进程然后舍弃，而是重复利用已经创建的进程。线程和进程在创建的时候就伴随着一定的开销，因此像这样先创建好再重复利用的技术是非常普遍的。这种重复利用的技术被称为池（pooling）（图 8）。

```
class HashFold
  class Pool ◀—— 用于进程池的类
    def initialize(hf, n) ◀—— 初始化,指定HashFold对象以及进程池中的进程数量
      pool = n.times.map{ ◀—— 创建n个进程
        c0,p0 = IO.pipe ◀—— 通信管道:从父进程到子进程(输入)
        p1,c1 = IO.pipe ◀—— 通信管道:从子进程到父进程(输出)
        fork do ◀—— 创建子进程
          p0.close ◀—— 关闭不使用的pipe
          p1.close
          loop do ◀—— 重复利用,执行循环
            input = Marshal.load(c0) rescue exit ◀—— 用Marshal等待输入,输入失败则exit
            hash = {} ◀—— 保存结果用的Hash
            hf.map(input) do |k,v| ◀—— 调用HashFold对象的nap方法
              hf.hash_merge(hash,k,v) ◀—— 数据保存,解决重复
            end
```

图 8　运用进程池的 HashFold

```
          Marshal.dump(hash,c1)  ◀——  将结果返回父进程
        end
      end
      c0.close  ◀——  父进程中也关闭不使用的pipe
      c1.close
      [p0,p1]  ◀——  对输入输出用的pipe进行map
    }
    @inputs = pool.map{|i,o| i}  ◀——  向进程池写入用的IO
    @outputs = pool.map{|i,o| o}  ◀——  由进程池读出用的IO
    @ichann = @inputs.dup  ◀——  可以向进程池写入的IO
    @queue = []  ◀——  写入队列
    @results = []  ◀——  读出队列
  end

  def flush  ◀——  将写入队列中的数据尽量多地写入
    loop do
      if @ichann.empty?
        o, @ichann, e = IO.select([], @inputs, [])  ◀——  使用select寻找可写的IO（a）
        break if @ichann.empty?  ◀——  如果没有可写的IO则放弃
      end
      break if @queue.empty?  ◀——  如果不存在要写入的数据则跳出循环
      Marshal.dump(@queue.pop, @ichann.pop)  ◀——  可写则执行写入
    end
  end

  private :flush  ◀——  这是一个用作内部实现的方法,因此声明为private
  def push(obj)  ◀——  向Pool写入数据的方法
    @queue.push obj
    flush
  end

  def fill  ◀——  从读出队列中尽量多地读出数据
    t = @results.size == 0 ? nil: 0       ◀——  result队列为空时用select阻塞,不为空时
则只检查(timeout=0)
    ochann, i, e = IO.select(@outputs, [], [], t)  ◀——  获取等待读出的IO（b）
    return if ochann == nil  ◀——  发生超时的时候
    ochann.each do
      c = ochann.pop
      begin
        @results.push Marshal.load(c)
      rescue => e
        c.close
        @outputs.delete(c)
      end
    end
  end
  private :fill  ◀——  用于内部实现的方法,因此声明为private

  def result
    fill  ◀——  从Pool中获取数据的方法
```

图 8　运用进程池的 HashFold（续）

```
      @results.pop
    end
  end

  def initialize(n=2)
    @pool = Pool.new(self,n)  ←— HashFold初始化,参数为构成池的进程数
  end  ←— 仅创建进程池

  def hash_merge(hash,k,v)
    if hash.key?(k)  ←— Hash合并
      hash[k] = self.fold(hash[k], v)
    else
      hash[k] = v
    end
  end

  def start(inputs)
    inputs.each do |input|  ←— HashFold计算开始
      @pool.push(input)  ←— 将各输入传递给Pool
    end

    hash = {}
    inputs.each do |input|
      @pool.result.each do |k,v|  ←— 获取结果用的Hash
        hash_merge(hash, k,v)  ←— 将结果Hash进行合并
      end
    end
    hash
  end
end
```

图 8　运用进程池的 HashFold（续）

和图 7 程序相比，由于增加了重复利用的代码，因此程序变得更复杂了。不过，要想象出这个程序的行为也并不难。

和图 7 程序相比，具体的区别在于并非每个输入都生成一个进程，而是实现启动一定数量的进程，对这些进程传递输入，再从中获取输出，如此反复。因此，图 7 的程序中只需要用一对 pipe，而这次的程序就需要分别用于输入和输出的两对 pipe。

此外，在并发编程中还有一点很重要，那就是不要发生阻塞。如果试图从一个还没有准备好数据的 pipe 中读取数据的话，在数据传递过来之前程序就会停止响应。这种情况被称为阻塞。

如果是非并行的程序，在数据准备好之前发生阻塞也是很正常的。不过，在并行程序中，在阻塞期间其他进程的输入也会停滞，从结果上看，完成处理所需要的时间就增加了。

因此，我们在这里用 select 来避免阻塞的发生。select 的参数是 IO 排列而成的数组，它可以返回数据已准备好的 IO 数组。select 可以监视读取、写入、异常处理 3 种数据，这次我们对

读取和写入各自分别调用 select。

图 8 的 (a) 处，对位于池中进程的写入检查，我们使用了 select。select 的参数是要监视的 IO 数组，但这里我们需要检查的只是写入，因此只在第 2 个参数指定了一个 IO 数组，第 1、第 3 参数都指定了空的数组。

图 8 的 (b) 处，我们对从进程池中读出结果进行检查。select 在默认情况下，当不存在可读出的 IO 时会发生阻塞，但当读出队列中已经有的数据时我们不希望它发生阻塞。因此我们指定了一个第 4 参数，也就是超时时间。select 的第 4 参数指定一个整数时，等待时间不会超过这个最大秒数。在这次的程序中，当队列不为空时我们指定了 0，也就是立即返回的意思。

要避免发生阻塞，除了 select 之外还有其他手段，比如使用其他线程。不过，一般来说，通过 fork 创建进程和线程不推荐在一个程序中同时使用，最大的理由是，pthread 和 fork 组合起来时，实际可调用的系统调用非常有限，因此在不同的平台上很难保证它总能够正常工作。

出于这个原因，同时使用 fork 和线程的程序，可能会导致 Ruby 解释器出现不可预料的行为。例如有报告说在 Linux 下可以工作，但在 FreeBSD 下则不行，这会导致十分棘手的 bug。

那么，我们用图 8 的 HashFold 来测试一下实际的运行速度吧。和之前其他程序一样在 1.9 的相同条件下运行，结果是 user 0.72 秒，system 0.06 秒，total 10.604 秒。由于不存在生成大量进程所带来的开销，性能有了稍许提升。此外，对抖动的抵抗力应该也提高了。顺便提一句，1.8.7 下的运行时间为 25.854 秒。

小结

对于我们这些老古董程序员来说，fork、pipe、select 等都是已经再熟悉不过的多进程编程 API 了，而这些 API 甚至可以用在最新的多核架构上面，真是感到无比爽快。

不过，目前市售的一般 PC，虽说是多核，但对于一台电脑来说也就是双核或者四核，稍微贵一些的服务器可以达到 8 核，而一台电脑拥有数十个 CPU 核心的超多核（many-core）环境还尚未成为现实。HashFold 等计算模型本来的目的是为了应对信息爆炸，而以目前这种程度的 CPU 核心数量，尚无法应对信息爆炸级别的数据处理。

看来今后我们必须要更多地考虑多台计算机构成的分布式环境了。

4.4 进程间通信

在有限的时间内处理大量的信息，其基本手段就是“分割统治”。也就是说，关键是如何分割大量的数据并进行并行处理。在并行处理中，充分利用多核（一台电脑具备多个 CPU）和分布式环境（用多台计算机进行处理）就显得非常重要。

进程与线程

并行处理的单位，大体上可以分为进程和线程两种（表 1）。

表1　处理的单位和同时运行的特征

处理的单位	内存空间共享	利用多核
线程	是	因实现不同而不同
进程	否	是

进程指的是正在运行的程序。各进程是相互独立的，用一般的方法，是无法看到和改变其他进程的内容和状态的。在 Linux 等 UNIX 系操作系统中，进程也无法中途保存状态或转移到另一台计算机上。即便存在让这种操作成为可能的操作系统，也只是停留在研究阶段而已，目前并没有民用化的迹象。

另一方面，多个线程可以在同一个进程中运行，线程间也可以相互合作。所属于同一个进程的各线程，其内存空间是共享的，因此，多个线程可以访问相同的数据。这是一个优点，但同时也是一个缺点。

说它是优点，是因为可以避免数据传输所带来的开销。在各进程之间，内存是无法共享的，因此进程间通信就需要对数据进行拷贝，而在线程之间进行数据共享，就不需要进行数据的传输。

而这种方式的缺点，就是由于多个线程会访问相同的数据，因此容易产生冲突。例如引用了更新到一半的数据，或者对相同的数据同时进行更新导致数据损坏等，在线程编程中，由于操作时机所导致的棘手 bug 是肯定会遇到的。

虽然灵活使用线程是很重要的，但总归线程的使用范围是在一台计算机中，而大规模的数据仅靠一台计算机是无法处理的。在这一节中，我们主要来介绍一下多台计算机环境中的进程间通信。

同一台计算机上的进程间通信

首先，我们来看同一台计算机上的进程间通信。正如我们在 4-3 中讲过的 HashFold 的实现，在同一台计算机上充分利用多个进程可以带来一定的好处。尤其是在现在的 Ruby 实现中，由于技术上的障碍使得靠线程来利用多核变得很困难（JRuby 除外），因此对进程的活用也就变得愈发重要了。

在 Linux 等 UNIX 系操作系统中，同一台计算机上进行进程间通信的手段有以下几种：

- 管道（pipe）
- 消息（message）
- 信号量（semaphore）
- 共享内存
- TCP 套接字
- UDP 套接字
- UNIX 域套接字

我们从上到下依次解释一下。管道是通过 pipe 系统调用创建一对文件描述符来进行通信的方式。所谓文件描述符，就是表示输入输出对象的一种识别符，在 Ruby 中对应了 IO 对象。当数据从某个 pipe 写入时，可以从另一端的 pipe 读出。事先将管道准备好，然后用“fork”系统调用创建子进程，这样就可以实现进程间通信了。

消息、信号量和共享内存都是 UNIX 的 System V(5) 版本中加入的进程间通信 API。其中消息用于数据通信，信号量用于互斥锁，共享内存用于在进程间共享内存状态。它们结合起来被称为 sysvipc。

不过，上述这些手段都不是很流行。例如管道的优点在于非父子关系的进程之间也可以实现通信，但是当不再使用时必须显式销毁，否则就会一直占用操作系统资源。说实话这并不是一个易用的 API，而关于它的使用信息又很少，于是就让人更加不想去用了，真是一个恶性循环。

套接字（socket）指的是进程间通信的一种通道。它原本是 4.2BSD 中包含的一个功能，但除了 UNIX 系操作系统之外，包括 Windows 在内的各种其他操作系统都提供了这样的功能。

套接字根据通信对象的指定方法以及通信的性质可以分为不同的种类，其中主要使用的包括 TCP 套接字、UDP 套接字和 UNIX 域套接字三种。它们的性质如表 2 所示。

使用套接字进行通信，需要在事先设定好的连接目标处，通过双方套接字的相互连接创建一个通道。这个连接目标的指定方法因套接字种类而异，在使用最多的 TCP 套接字和 UDP 套

接字中，是通过主机地址（主机名或者 IP 地址）和端口号（1 到 65535 之间的整数）的组合来指定的。

表2　套接字的分类与特征

套接字种类	连接目标指定方法	数据分隔	可靠性
TCP套接字	主机地址+端口号	不保存	有
UDP套接字	主机地址+端口号	保存	无
UNIX域套接字	路径	保存	有

位于网络中的每台计算机，都拥有一个被称为 IP 地址的识别码（IPv4 是 4 字节的序列，IPv6 是 16 字节的序列）。例如在 IPv4 中，自己正在使用的电脑所对应的 IP 地址为“127.0.0.1”①。在开始通信时，通过指定对方计算机的 IP 地址，就相当于决定了要和哪台计算机进行通信。

IP 地址是一串数字，非常难记，因此每台计算机都还有一个属于自己的“主机名”。在这里就不讲述或多细节了，不过简单来说，通过 DNS（Domain Name System，域名系统）这一机制就可以由主机名来获得 IP 地址了。

另一方面，UNIX 域套接字则是使用和文件一样的路径来指定连接目标。在服务器一端创建监听用的 UNIX 域套接字时，需要指定一个路径，而结果就是将 UNIX 域套接字创建在这个指定的路径中。

以路径作为连接目标，就意味着 UNIX 域套接字只能用于同一台计算机上的进程间通信。不过，UNIX 域套接字还具备一些特殊的功能，它不仅可以传输一般的字节流，还可以传输文件描述符。

TCP 套接字被称为流套接字（stream socket），写入的数据只能作为单纯的字节流来对待，因此无法保存每次写入的数据长度信息。

相对地，UDP 套接字和 UNIX 流套接字中，写入的数据是作为一个包（数据传输的单位）来发送的，因此可以获取每次写入的数据长度。不过，当数据过长时，数据包会根据各操作系统所设定的最大长度进行分割。

对于 UDP 套接字，有一点需要注意，那就是基于 UDP 套接字的通信不具备可靠性。所谓没有可靠性，就是说在通信过程中可能会发生数据到达顺序颠倒，最坏的情况下，还可能发生数据在传输过程中丢失的情况。

① IPv6 中本机地址表示为“::1”，关于 IPv6 的细节在本书中就不再赘述了。（原书注）

TCP/IP 协议

利用网络进行通信的协议（protocol）迄今为止已经出现了很多种，但其中一些因为各种原因已经被淘汰了，现在依然幸存下来的就是一种叫做 TCP/IP 的协议。TCP 套接字就是"用 TCP 协议进行通信的套接字"的意思。

TCP 是 Transmission Control Protocal（传输控制协议）的缩写。TCP 是负责错误修恢复、数据再发送、流量控制等行为的高层协议，它是在一种更低层级的 IP 协议（即 Internet Protocol）的基础之上实现的①。

UDP 则是 User Datagram Protocol（用户数据报协议）的缩写。UDP 实际上是在 IP 的基础上穿了一件薄薄的马甲，和 TCP 相比，它有以下这些不同点。

1. 保存通信数据长度

在 TCP 中，发送的数据是作为字节流来处理的。虽然在实际的通信过程中，数据流会被分割为一定大小的数据包，但在 TCP 层上这些包是连接在一起的，无法按照包为单位来查看数据。

相对地，通过 UDP 发送的数据会直接以数据包为单位进行发送，作为发送单位的数据包长度会一直保存到数据接收方。不过，如果包的长度超过操作系统所规定的最大长度（一般为 9KB 左右）就会被分割开，因此也无法保证总是能获取原始的数据长度。

2. 没有纠错机制

要发送的数据在经过网络到达接收方的过程中，可能会发生一些状况，比如数据包的顺序发生了调换，最坏的情况下甚至发生整个数据包丢失。在 TCP 中，每个数据包都会被赋予一个编号，如果包顺序调换，或者本来应该收到的包没有收到，接收方会通知发送方"某个编号的包没有收到"，并请求发送方重新发送该包，这样就可以保证数据不会发生遗漏。

此外，还可以在网络繁忙的时候，对一次发送数据包的大小和数量进行调节，以避免网络发生阻塞。

相对地，UDP 则没有这些机制，像"顺序调换了"、"发送的数据没收到"这样的情况，必须自己来应付。

---

① 网络通信协议从低到高共分为 5 层（分别对应 OSI 通信模型中的第 3 ～ 7 层），其中 TCP 和 UDP 都属于较高的"传输层"（OSI 第 6 层），IP 属于更低一级的"网络层"（OSI 第 5 层），而更常见的 HTTP、FTP 等则属于最高的"应用层"（OSI 第 7 层）。

3. 不需要连接

在 TCP 中，通信对象是固定的，因此，如果要和多个对象进行通信，则需要对每个对象分别使用不同的套接字。

相对地，UDP 则是使用 sendto 系统调用来显式指定发送目标，因此每次发送数据时可以发送给不同的目标。在接收数据时，也可以使用 recvfrom 系统调用，一并获取到发送方的信息。虽然 UDP 不需要进行连接，但在需要的情况下，也可以进行显式的连接来固定通信对象。

4. 高速

由于 TCP 可以进行复杂的控制，因此使用起来比较方便。但是，由于需要处理的工作更多，其实时性便打了折扣。

UDP 由于处理工作非常少，因而能够发挥 IP 协议本身的性能。在一些实时性大于可靠性的网络应用程序中，很多是出于性能上的考虑而选择了 UDP。

例如，在音频流的传输中，即便数据发生丢失也只不过是造成一些音质损失（例如产生一些杂音）而已。相比之下，维持较低的延迟则显得更加重要。在这样的案例中，比较适合采用 UDP 协议来进行通信。

用 C 语言进行套接字编程

在套接字的使用上，已经有了用系统调用构建的 C 语言 API。通过 C 语言可以访问的套接字相关系统调用如表 3 所示。TCP 套接字的使用方法和步骤，以及无连接型 UDP 的步骤如图 1 所示。

表3　套接字相关系统调用

系统调用	功　　能
accept（fd, addr, len）	接受连接并返回一个新的套接字
bind（fd, addr, len）	对服务器端套接字命名
connect（fd, addr, len）	套接字连接
getsockopt（fd,level,optname,optval,optlen）	获取套接字选项
listen（fd, n）	设置连接队列（固定用法）
recv（fd, data, len, flags）	接收数据（可指定 flags）
recvfrom（fd, data, len, flags, addr, alen）	包含发送方信息的数据接收
send（fd, data, len, flags）	发送数据（可指定 flags）
sendto（fd, data, len, flags, addr, alen）	指定接收方的数据发送
setsockopt（fd,level,optname,optval,optlen）	设置套接字选项
socket（domain, type, protocol）	创建套接字

图 2 是一个使用套接字相关系统调用进行套接字通信的客户端程序。这个程序访问本机（localhost）的 13 号端口，将来自该端口的数据直接输出至标准输出设备。

13 号端口是返回当前时间的“daytime”服务端口号码，所返回的当前时间是一个字符串。最近的操作系统倾向于关闭所有不必要的服务，因此 daytime 服务可能不可用。如果你电脑上的 daytime 服务正常工作的话，运行这个程序将显示类似下面这样的字符串：

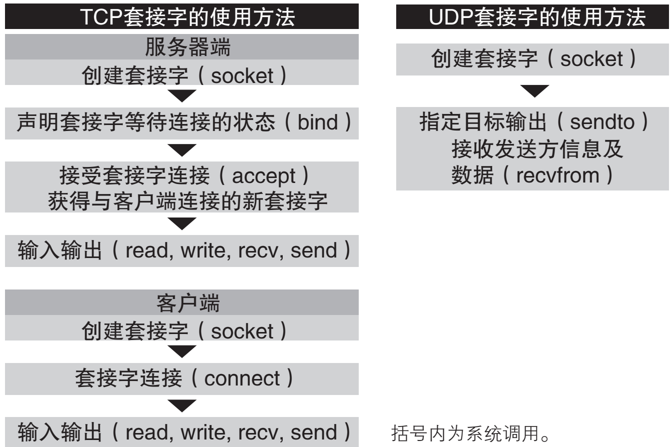


图 1　连接型 TCP 套接字和无连接型 UDP 套接字的使用方法

```
Sat Oct 10 15:26:28 2009
```

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int main()
{
    int sock;
    struct sockaddr_in addr;
    struct hostent *host;
    char buf[128];
    int n;

    sock = socket(PF_INET, SOCK_STREAM, 0);  ◄── socket系统调用
    addr.sin_family = AF_INET;                                       ┐
    host = gethostbyname("localhost");                               │
    memcpy(&addr.sin_addr, host->h_addr, sizeof(addr.sin_addr));     │ 指定连接目标
    addr.sin_port = htons(13);  /* daytime service */                ┘
    connect(sock, (struct sockaddr*)&addr, sizeof(addr));
    n = read(sock, buf, sizeof(buf));
    buf[n] = '\0';
    fputs(buf, stdout);
}
```

图 2　用 C 语言编写的网络客户端

用 C 语言来编写程序，仅仅是打开套接字并读取数据这么简单的操作，也需要十分繁琐的代码。

那我们就来看一看程序的内容吧。首先通过第 15 行的 socket 系统调用创建套接字。其中参数的意思是使用基于 IP 协议的流连接（TCP）(表 4)。第 3 个参数“0”表示使用指定通信域的“最普通”的协议，一般情况下设为 0 就可以了。

表4　socket系统调用的参数

协议类型	说　明
PF_INET	IPv4 协议
PF_INET6	IPv6 协议
PF_APPLETALK	ApleTalk 协议
PF_IPX	IPX 协议
通信类型	
SOCK_STREAM	字节流套接字
SOCK_DGRAM	数据报套接字

第 16 ～ 19 行用于指定连接目标。sockaddr_in 结构体中存放了连接目标的地址类型（类别）、主机地址和端口号。

需要注意的是第 19 行中指定端口号的 htons()。htons() 函数的功能是将 16 位整数转换为网络字节序（network byte order），即各字节的发送顺序。由于套接字连接目标的指定全部需要通过网络字节序来进行，如果忘记用这个函数进行转换的话就无法正确连接。

服务器端程序则更加复杂，因此在这里不再赘述，不过大家应该对用 C 语言处理网络连接有一个大概的了解了吧。

用 Ruby 进行套接字编程

以系统调用为基础的 C 语言套接字编程相当麻烦。那么，Ruby 怎么样呢？图 3 是和图 2 的 C 语言程序拥有相同功能的 Ruby 程序。

```
require 'socket'
print TCPSocket.open("localhost", "daytime").read
```

图 3　Ruby 编写的网络客户端

值得注意的是，除了库引用声明“require”那一行之外，实质上只需要一行代码就完成了套接字连接和通信。和 C 语言相比，Ruby 的优势相当明显。

用套接字进行网络编程是 Ruby 最擅长的领域之一，原因如下。

1. 瓶颈

在程序开发中，对于是否采用 Ruby 这样的脚本语言，犹豫不决的理由之一就是运行性能。

在比较简单的工作中，如果由于解释器的实现方式导致性能下降，其影响是相当大的。如果用一个简单的循环来测试程序性能，那么 Ruby 程序速度可能只有 C 语言程序的十分之一甚至百分之一。光从这一点来看，大家不禁要担心，这么慢到底能不能用呢？

不过，程序的运行时间其实大部分都消耗在 C 语言编写的类库内部，对于拥有一定规模的实用型程序来说，差距并没有那么大。

更进一步说，对于以网络通信为主体的程序来说，其瓶颈几乎都在于通信部分。和本地访问数据相比，网络通信的速度是非常慢的。既然瓶颈在于通信本身，那么其他部分即便运行速度再快，也和整体的性能关系不大了。

2. 高级 API

C 语言中可以使用的套接字 API 包括结构体和多个系统调用，非常复杂。

在图 2 的 C 语言程序中，为了指定连接目标，必须初始化 sockaddr_in 结构体，非常麻烦。相对地，在 Ruby 中由于 TCPSocket 类提供了比较高级的 API，因此程序可以变得更加简洁易懂。如果想和 C 语言一样使用套接字的全部功能，通过支持直接访问系统调用的 Socket 类就可以实现了。

Ruby 的套接字功能

那么，我们来详细看看 Ruby 的套接字功能吧。在 Ruby 中，套接字功能是由“socket”库来提供的。要使用 socket 库的功能，需要在 Ruby 程序中通过下面的方式来加载这个库：

```
require 'socket'
```

socket 库中提供的类包括 BasicSocket、IPSocket、TCPSocket、TCPServer、UDPSocket、UNIXSocket、UNIXServer 和 Socket（图 4）。在客户端编程上，恐怕其中用得最多的应该是 TCPSocket，而在服务器端则是 TCPServer。

```
IO
└── BasicSocket
     ├── IPSocket
     │    ├── TCPSocket
     │    │    └── TCPServer
     │    └── UDPSocket
     ├── UNIXSocket
     │    └── UNIXServer
     └── Socket
```

图 4　套接字相关的类

其中 Socket 类可以调用操作系统中套接字接口的所有功能，但由于是直接访问操作系统的接口，因此程序就会变得比较复杂。Ruby 的套接字属于 IO 的子类，因此对套接字也可以进行普通的输入输出操作，这一点非常方便。

BasicSocket 是 IO 的直接子类，同时也是其他所有套接字类的超类。BasicSocket 是一个抽象类，并不能创建实例。BasicSocket 类中的方法如表 5 所示。

IPSocket 是 BasicSocket 的子类，也是 TCPSocket、UDPSocket 的超类，它包含了这两个类

共通的一些功能，也是一个抽象类。IPSocket 类中的方法如表 6 所示。

表5　BasicSocket类的方法

实例方法	说　　明
close_read	关闭读取
close_write	关闭写入
getpeername	连接目标套接字信息
getsockname	自己的套接字信息
getsockopt（opt）	获取套接字选项
recv（len [,flag]）	数据接收
send（str [,flag]）	数据发送
setsockopt（opt,val）	设置套接字选项
shutdown（[how]）	结束套接字通信

表6　IPSocket类的方法

实例方法	说　　明
addr	自己的套接字信息
peeraddr	连接目标套接字信息
recvfrom（len [,flag]）	数据接收

TCPSocket 是连接型套接字，即和通信对方进行连接并进行连续数据传输的套接字。TCPSocket 是一个具体类（可以直接创建实例的类）。创建实例需要使用 new 方法，new 方法的调用方式为 new (host, port)，可以完成套接字的创建和连接操作。

TCPServer 是 TCPSocket 的服务器版本，通过这些类可以大大简化服务器端的套接字处理。当为 new 方法指定两个参数时，可以限定只接受来自第一个参数所指定的主机的连接（表 7）。

表7　TCPServer类的方法

类　方　法	说　　明
new（[host,] port）	套接字的创建和连接
实例方法	
accept	接受连接
listen（n）	设置连接队列

UDPSocket 是对 UDP 型套接字提供支持的类。UDP 型套接字是无连接型套接字，其特征是可以保存每次写入的数据长度。UDPSocket 类中的方法如表 8 所示。

表8　UDPSocket类的方法

类　方　法	说　　明
new（[socktype]）	创建套接字
实例方法	
bind（host,port）	为套接字命名
connect（host, port）	套接字连接
send（data [,flags,host, port]）	发送数据

UNIXSocket 是用于 UNIX 域套接字的类。UNIX 域套接字是一种用于同一台计算机上进程间通信的手段，在通信目标的指定上采用“文件路径”的方式，其他方面和 TCPSocket 相同，也是需要连接并进行流式输入输出。UNIXSocket 类中的方法如表 9 所示。

send_io 和 recv_io 这两个方法是 UNIX 域套接字的独门功夫。使用这两个方法，可以通过 UNIX 域套接字将文件描述符传递给其他进程。一般来说，在进程间传递文件描述符，只能通

过具有父子关系的进程间共享这一种方式，但使用 UNIX 域套接字就可以在非父子关系的进程间实现文件描述符的传递了。

UNIXServer 是 UNIXSocket 的服务器版本。和 TCPServer 一样，用于简化套接字服务器的实现。其中所补充的方法也和 TCPServer 相同。

最后要介绍的 Socket 类是一个底层套接字接口。Socket 类所拥有的方法对应着 C 语言级别的全部套接字 API，因此，只要使用 Socket 类，就可以和 C 语言一样进行同样细化的程序设计，但由于这样实在太繁琐所以实际上很少用到。Socket 类中的方法如表 10 所示，套接字相关各类的功能一览如表 11 所示。

表9　UNIXSocket类的方法

类　方　法	说　　明
new（path）	创建套接字
socketpair	创建套接字对
实例方法	
path	套接字路径
addr	自己的套接字信息
peeraddr	连接目标套接字信息
recvfrom（len［,flag］）	数据接收
send_io（io）	发送文件描述符
recv_io(［klass,mode］)	接收文件描述符

表10　Socket类的方法

类　方　法	说　　明
new（domain,type,protocol）	创建套接字
socketpair（domain,type,protocol）	创建套接字对
gethostname	获取主机名
gethostbyname（hostname）	获取主机信息
gethostbyaddr（addr, type）	获取主机信息
getservbyname（name[,proto]）	获取服务信息
getaddrinfo（host,service[,family,type,protocol]）	获取地址信息
getnameinfo（addr[,flags]）	获取地址信息
pack_sockaddr_in（host,port）	创建地址结构体
unpack_sockaddr_in（addr）	解包地址结构体
实例方法	
accept	等待连接
bind（addr）	为套接字命名
connect（host, port）	连接套接字
listen（n）	设置连接队列
recvfrom（len[,flag]）	数据接收

表11　套接字相关的类

BasicSocket	所有套接字类的超类（抽象类）
IPSocket	执行IP通信的抽象类
TCPSocket	连接型流套接字
TCPServer	TCPSocket用的服务器套接字
UDPSocket	无连接型数据报套接字
UNIXSocket	用于同一主机内进程间通信的套接字
UNIXServer	UNIXSocket用的服务器套接字
Socket	可使用Socket系统调用所有功能的类

用 Ruby 实现网络服务器

我们已经通过 C、Ruby 两种语言介绍了客户端套接字编程的例子，下面我们来看看服务器端的设计。刚才那个访问 daytime 服务的程序可能有很多人都无法成功运行，于是我们来编写一个和 daytime 服务器拥有相同功能的服务器程序。原来的 daytime 服务端口只能由 root 账号使用（1024 号以内的端口都需要 root 权限），因此我们将连接端口设置为 12345（图 5）。

```
require 'socket'
s = TCPServer.new(12345)
loop {
  cl = s.accept
  cl.print Time.now.strftime("%c")
  cl.close
}
```

图 5　Ruby 编写的网络服务器

这样就完成了。网络服务器可能给人的印象很庞大，其实却出人意料地简单。这也要归功于 TCPServer 类所提供的高级 API。

先运行这个程序，然后从另一个终端窗口中运行刚才的客户端程序（C 语言版见图 2，Ruby 版见图 3），运行之前别忘了将 daytime 的部分替换成“12345”。运行结果如果显示出类似下面这样的一个时间就表示成功了。

```
Mon Jun 12 18:52:38 2006
```

下面我们来简单讲解一下图 5 的这个程序。第 2 行我们创建了一个 TCPServer，参数是用于连接的端口号，仅仅如此我们就完成了 TCP 服务的建立。

第 3 行开始是主循环。第 4 行中对于 TCPServer 套接字调用 accept 方法。accept 方法会等待来自客户端的连接，如果有连接请求则返回与客户端建立连接的新套接字，我们在这里将新套接字赋值给变量 cl。客户端套接字是 TCPSocket 的对象，即 IO 的子类，因此它也是一个可以执行一般输入输出操作的对象。

第 5 行 print 当前时间，daytime 服务的处理就这么多了。处理完成后将客户端套接字 close 掉，然后调用 accept 等待下一个连接。

图 5 的程序会对请求逐一进行处理。对于像 daytime 这样仅仅是返回一个时间的服务也许还好，如果是更加复杂的处理的话，这样可就不行了。如果 Web 服务器在完成前一个处理之前无法接受下一个请求，其处理性能就会下降到无法容忍的地步。在这样的情况下，使用线程或进程进行并行处理是比较常见的做法。使用线程进行并行化的程序如图 6 所示。

```
require 'socket'
s = TCPServer.new(12345)
loop {
  Thread.start(s.accept) { |cl|
    cl.print Time.now.strftime("%c")
    cl.close
  }
}
```

图 6　用线程实现并行处理的程序

正如大家所见，用 Ruby 进行网络编程是非常容易的。有很多人认为提到 Ruby 就必然要提到 Web 编程，或许不如说，只有网络编程才能发挥 Ruby 真正的价值吧。

小结

利用套接字，我们就可以通过网络与地球另一端的计算机进行通信。不过，套接字所能传输的数据只是字节序列而已，如果要传输文本以外的数据，在传输前需要将数据转换为字节序列。

这种转换一般称为序列化（serialization）或者封送（marshaling）。在分布式编程环境中，由于会产生大量数据的传输，因此序列化通常会成为左右整体性能的一个重要因素。

4.5 Rack 与 Unicorn

Web 应用程序服务器主要由 HTTP 服务器与 Web 应用程序框架构成。说起 HTTP 服务器，Apache 是很有名的一个，但除此之外还有其他很多种，例如高性能的新型轻量级服务器 nginx、以纯 Ruby 实现并作为 Ruby 标准组件附带的 WEBrick，以及以高速著称的 Mongrel 和 Thin 等。

此外，Web 应用程序框架方面，除了鼎鼎大名的 Ruby on Rails 之外，还出现了如 Ramaze、Sinatra 等“后 Rails”框架。于是，对于 Web 框架来说，就必须要对所有的 HTTP 服务器以及应用程序服务器提供支持，这样的组合方式真可为多如牛毛。

为了解决如此多的组合，出现了一种叫 Rack 的东西（图 1）。Rack 是在 Python 的 WSGI[①] 的影响下开发的用于连接 HTTP 服务器与框架的库。HTTP 服务器一端，只需对 Rack 发送请求，然后接受响应并进行处理，就可以连接所有支持 Rack 的框架。同样地，在框架一端也只需要提供对 Rack 的支持，就可以支持大多数 HTTP 服务器了。最近以 Ruby 为基础的 Web 应用程序框架，包括 Rails 在内，基本上都已经支持 Rack 了。

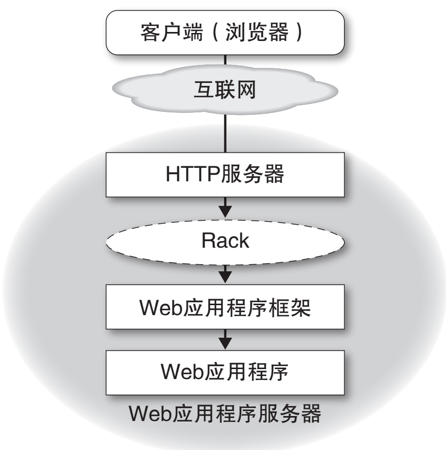


图 1 Web 应用程序服务器架构

Rack 的基本原理，是将对 Rack 对象发送 HTTP 请求的“环境”作为参数来调用 call 方法，并将以返回值方式接收的请求组织成 HTTP 请求。Rack 的 Hello World 程序如图 2 所示。

```
class HelloApp
  def call(env)
    [200, {"Content-Type" => "text/plain"},
    ["Hello, World"]]
  end
end
```

图 2 Hello World Rack 应用程序

① WSGI：Web Server Gateway Interface（Web 服务器网关接口）的缩写，是 Python 中使用的一种 HTTP 服务器与 Web 应用程序框架之间的通用接口。（原书注）

Rack 对象所需的要素包括下面两个：

- 带有一个参数的 call 方法。call 方法在调用时，其参数为表示请求环境的 Hash。
- call 方法返回带 3 个元素的数组。第 1 个元素为状态代码（整数），第 2 个元素为表示报头的 Hash，第 3 个元素为数据本体（字符串数组）。

像 Rack 专用类等特殊的数据结构是不需要的。

好了，为了看看 Rack 应用程序实际是如何工作的，我们将图 2 的程序保存到“hello.rb”这个文件中，并另外准备一个名为“hello.ru”的配置文件（图 3）。hello.ru 虽然说是一个配置文件，但其实体只是一个单纯的 Ruby 程序而已。准备好 hello.ru 文件之后，我们就可以使用 Rack 应用程序的启动脚本“rackup”来启动应用程序了。

```
require 'rubygems'
require 'rack'
require 'hello'

run HelloApp.new
```

图 3　配置文件 hello.ru

```
$ rackup hello.ru ⏎
```

然后，我们只要用 Web 浏览器访问 http://localhost:9292/，就会显示出 Hello World 了。这次我们都用了默认配置，端口号为“9292”，HTTP 服务器则是用了“WEBrick”，但通过配置文件是可以修改这些配置的。

Rack 中间件

Rack 的规则很简单，就是将 HTTP 请求作为环境对象进行 call 调用，然后再接收响应。因此，无论是 HTTP 服务器还是框架都可以很容易地提供支持。

应用这样的机制，只要在实际对框架进行调用之前补充相应的 call，就可以在不修改框架的前提下，对所有支持 Rack 的 Web 应用程序增加具备通用性的功能。这种方式被称为“Rack 中间件”。

Rack 库中默认自带的 Rack 中间件如表 1 所示。

中间件的使用可以通过在“.ru”文件中用“use”来进行指定。例如，如果要对 Web 应用添加显示详细日志、产生异常时声称生成错误页面以及显示错误状态页面的功能，可以将图 3 的 hello.ru 文件改写成图 4 这样。每个功能的添加只需要一行代码就可以完成，可见其表述力非常优秀。

表1　Rack中间件

中　间　件	内　　容
Rack::Auth::Basic	BASIC认证
Rack::Auth::Digest	Digest认证
Rack::Auth::OpenID	OpenID认证
Rack::Reloader	当Ruby脚本更新时重新加载
Rack::File	显示静态文件
Rack::Cascade	组合Web应用，找不到文件时尝试下一个
Rack::Static	指定目录显示静态文件
Rack::Lint	检查应用是否符合Rack规范（开发用）
Rack::ShowExceptions	由应用产生的异常生成错误页面
Rack::ShowStatus	生成状态码400、500用的错误页面
Rack::CommonLogger	生成Apache common log格式的日志
Rack::Recursive	用异常进行跳转
Rack::Session::Cookie	用cookie进行会话管理
Rack::Session::Pool	用会话ID进行会话管理

```
require 'rubygems'
require 'rack'
require 'hello'

use Rack::CommonLogger
use Rack::ShowExceptions
use Rack::ShowStatus

run HelloApp.new
```

图4　hello.ru（使用中间件）

应用程序服务器的问题

正如上面所讲到的，只要使用Rack，HTTP服务器与Web框架就可以进行自由组合了。这样一来，我们可以根据情况选择最合适的组合，但如果网站的流量达到一定的规模，更常见的做法是将Apache和nginx放在前端用作负载均衡，而实际的应用程序则通过Thin和Mongrel进行工作（图5）。

其中，Apache（或者nginx）负责接收来自客户端的请求，然后将请求按顺序转发给下属的Thin服务器，从而充分利用I/O等待等情况所产生的空闲时间。此外，最近的服务器大多都安装了多核CPU，像这样用多个进程分担工作的架构则可以最大限度地利用多核的性能。

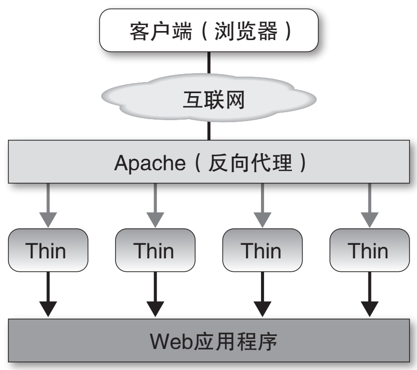


图5　Web应用程序架构

Thin是一种十分快速的HTTP服务器，在大多数情况下，这样的架构已经足够了。但在某些情况下，这种架构也会发生下面这些问题。

❑ 响应缓慢

- ❑ 内存开销
- ❑ 分配不均衡
- ❑ 重启缓慢
- ❑ 部署缓慢

下面我们来具体看看这些问题的内容。

1. 响应缓慢

由于应用程序的 bug，或者数据库的瓶颈等原因，应用程序的响应有时候会变得缓慢。虽然这是应用方面的问题，HTTP 服务器是没有责任的，但这样的问题导致超时是成为引发更大问题的元凶。

为了避免这样的问题对其他的请求产生过大的负面影响，默认情况下 Thin 会停止 30 秒内没有响应的任务并产生超时。但不幸的是，不知道是不是 Ruby 在线程实现上的关系，这个超时的机制偶尔会失灵。

2. 内存开销

有些情况下，负责驱动 Web 应用程序的 Thin 等服务器程序的内存开销会变得非常大。这种内存开销的增加往往是由于数据库连接未释放，或者垃圾回收机制未能回收已经死亡的对象等各种原因引发的。

无论如何，服务器上的内存容量总归是有限的，如果内存开销产生过多的浪费，就会降低整体的性能。

和响应缓慢一样，内存开销问题也可能会引发其他的问题。内存不足会导致处理负担增加，处理负担增加会导致其他请求响应变慢，响应变慢又会导致用户不断尝试刷新页面，结果让情况变得更加糟糕。一旦某个环节出现了问题，就会像“多米诺骨牌”一样产生连锁反应，这样的情况不算少见。

3. 分配不均衡

用 Apache 或 nginx 作为反向代理[①]，对多个 Thin 服务器进行请求分配的情况下，请求会由前端服务器按顺序转发给下属的 Thin 服务器。这种形态很像是上层服务器对下层的“发号施令”，因此又被称为推模式（push model）。

---

① 反向代理（reverse proxy）是指将代理服务器配置在 Web 服务器一侧的网络中，实现 Web 服务器的缓存、安全性、负载均衡等功能的技术，也可以指配置了这种技术的服务器本身。（原书注）

一般来说，在推模式下，HTTP 服务器将请求推送给下属服务器时，并不知道目标服务器的状态。原则上说，HTTP 服务器只是按顺序依次将请求转发给下属各服务器而已。

然而，当请求转发目标的服务器由于某些原因没有完成对前一个请求的处理时，被分配给这个忙碌服务器的请求就只能等待，而且前一个请求也不知道什么时候才能处理完毕，只能自认倒霉了。

4. 重启缓慢

像上面所说的情况，一旦对请求的处理发生延迟，负面影响就会迅速波及到很大的范围。当由于某些原因导致处理消耗的时间过长时，必须迅速对服务器进行重启。虽然 Thin 自带超时机制，但对于内存开销，以及基于 CPU 时间进行服务器状态监控，需要通过 Monit、God 等监控程序来完成。

这些程序会监控服务器进程，当发现问题时将进程强制停止并重新启动，但即便如此，重启服务依然不是一件简单的事。当监控程序注意到请求处理的延迟时，马上重启服务器进程，这时，框架和应用程序的代码需要全部重新加载，恢复到可以进行请求处理的状态至少需要几秒钟的时间。而在这段时间中，如果有哪个倒霉的请求被分配到这个正在重启的服务器进程，就又不得不进行长时间的等待了。

5. 部署缓慢

当需要对 Web 应用程序进行升级时，就必须重启目前正在运行的所有应用程序服务器。正如刚才所讲到的，仅仅是重启多个服务器进程中的一个，就会殃及到一些不太走运的请求，如果重启所有的服务器进程的话，影响就会更大。哪怕 Web 应用整体仅仅停止响应 10 秒钟，对于访问量很大的网站来说，也会带来超乎想象的损失。

有一种说法指出，对于网站从开始访问到显示出网页之间的等待时间，一般用户平均可以接受的极限为 4 秒。由于这个时间是数据传输的时间和浏览器渲染 HTML 时间的总和，因此 Web 应用程序用于处理请求的时间应尽量控制在 3 秒以内。

如果上述说法成立的话，那么目前这种在前端配置一个反向代理，并将请求推送给下属服务器的架构，虽然平常没有问题，但一旦发生问题，其负面影响就很容易迅速扩大，这的确可以说是一个缺点。

于是我们下面要介绍的，就是一个面向 UNIX 的 Rack HTTP 服务器——Unicorn。Unicorn 是以解决上述问题为目标而开发的高速 HTTP 服务器。之所以说是“面向 UNIX”的，是因为 Unicorn 使用了 UNIX 系操作系统所提供的 fork 系统调用以及 UNIX 域套接字，因此无法在 Windows 上工作。

Unicorn 的架构

Unicorn 系统架构如图 6 所示。这张图上使用的是 Apache，换成 nginx 也是一样的。

Unicorn 和图 5 中采用的 Thin 在架构上的区别在于：Apache 只需要通过 UNIX 域套接字和单一的 Master 进行通信。

在采用 Thin 的架构中，Apache 负责负载均衡，为下属各服务器分配请求，而在采用 Unicorn 的架构中，Apache 只需要和一个称为 Master 的进程进行通信即可。这种通信是通过 UNIX 域套接字来完成的。

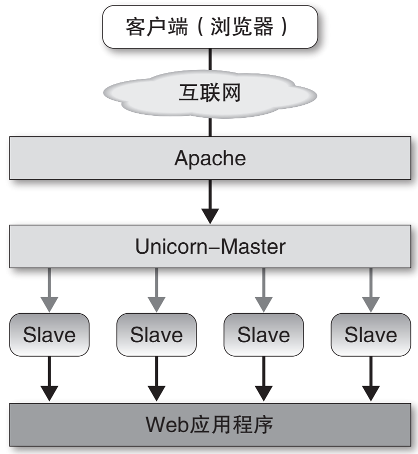


图 6　Web 应用程序架构

一般的套接字都是通过主机名和端口号来指定通信对象，而 UNIX 域套接字则是通过路径来指定的。服务器端（数据的接收方）通过指定一个路径来创建 UNIX 域套接字时，在指定的路径就会生成一个特殊的文件。开始通信的一方只要像一般文件一样写入数据，在接收方看来就像是在通过套接字来进行通信一样。

UNIX 域套接字具有一些方便的特性：①客户端可以像文件一样来进行读写操作；②进程之间不具备父子关系也可以进行通信。不过它也有缺点，由于通信对象的指定是采用路径的形式，因此只能用于同一台主机上的进程间通信。

然而，对于多台主机的分布式环境，也有通过 Unicorn 进行负载均衡的需求，这种情况下也可以用 TCP 套接字来代替 UNIX 域套接字，虽然性能会有一定的下降。

由 Apache 转发给 Unicorn-Master 的请求，会转发给由 Master 通过 fork 系统调用启动的 Slave，而实际的处理会在 Slave 中完成。然后，Master 会在 Slave 处理完成之后，将响应转发给 Apache。

Unicorn 的解决方案

不过，这样的机制如何解决 Thin 等所遇到的问题呢？对于上面提到的那 5 个问题，我们逐一来看一看。

1. 响应缓慢

Web 应用响应慢，本质上说还是应用自身的问题，因此无法保证一定能够避免。对于服务器来说，重要的是，当问题出现时如何避免波及到其他无关的请求。

在简单的推模式中，转发请求的时候，并不会向被分配到请求的服务器确认其是否已经完成对上一个请求的处理，因此导致对其他无关请求的处理发生延迟。

相对地，在 Unicorn 中，完成处理的 Slave 会主动去获取请求，即拉模式（pull model），因此从原理上说，不会发生某个请求被卡死在一个忙碌的服务器进程中的情况。

不过，即便是在拉模式下，也并非完全不存在请求等待的问题。当访问量超出了 Slave 的绝对处理能力时，由于没有空闲的 Slave 能够向 Master 索取请求，于是新来的请求便不得不在 Master 中进行等待了。

如果由于某种原因导致 Slave 完全停止运行的情况下，由于可用的 Slave 少了一个，整体的处理能力也就随之下降了。在 Unicorn 中，对于这样的情况所采取的措施是，当发现某个 Slave 的处理超出规定时间则强制重启该 Slave。

2. 内存开销

和响应缓慢的问题一样，内存的消耗也会影响到其他的请求。因此，当发生问题时，最重要的是如何在不影响其他请求的前提下完成重启。由于 Unicorn 可以快速完成对 Slave 的重启，因此在可以比较轻松地应对内存消耗的问题，理由我们稍后再介绍。

3. 分配不均

正如之前所讲过的，在采用拉模式的 Unicorn 中，不会发生将请求分配给不可用的服务器进程的问题。在 Unicorn 中，来自 Apache 的请求会通过 UNIX 域套接字传递给单一的 Unicorn-Master，而下属的 Slave 会在自己的请求处理完成之后向 Master 索取下一个请求。综上所述，在 Unicorn 中不会发生分配不均的问题。

4. 重启缓慢

采用拉模式来避免分配不均是 Unicorn 的一大优点，而另一大优点就是它能够快速重启。

Unicorn 对 Slave 进行重启时有两个有利因素。第一，由于采用了拉模式，因此即便重启过程中某个 Slave 无法工作，也不用担心会有任何请求分配到该 Slave 上，这样一来，整体上来看就不会发生处理的停滞。

第二，Unicorn 在 Slave 的启动方法上很有讲究，使得实际重启所花费的时间因此得以大大缩短。

当由于超时、内存开销过大等原因被监控程序强制终止，或者由于其他原因导致 Slave 进程停止时，Master 会注意到 Slave 进程停止工作，并立即通过 fork 系统调用创建自身进程的副本，并使其作为新 Slave 进程来启动。由于 Unicorn-Master 在最开始启动时就已经载入了包括框架和应用程序在内的全部代码，因此在启动 Slave 时只需要调用 fork 系统调用，并将 Slave 用的处理切换到子进程中就可以了。

在最近的 UNIX 系操作系统中，都具备了“Copy-on-Write”（写时复制）[①]功能，从而不需要在复制进程的同时复制内存数据。只需要在内核中重新分配一个表示进程的结构体，该进程所分配的内存空间就可以与调用 fork 系统调用的父进程实现共享。随着进程的执行，当实际发生对内存数据的改写时，仅将发生改写的内存页进行复制，也就是说，对内存的复制是随着进程执行的过程而循序渐进的，这样一来，基本上就可以避免因内存复制的延迟导致的 Slave 启动开销。

Thin 等应用程序服务器的重启过程则更为复杂。首先，需要启动 Ruby 解释器，然后还要重新载入框架和应用程序代码。相比之下，运用了 Unicorn 系统中的 Slave 的重启时间要短得多，这样一来，就可以毫不犹豫地重启发生问题的 Slave。此外，由于恢复工作可以快速完成，也可以避免系统整体响应产生延迟。

5. 部署缓慢

Slave 重启的速度很快，也就意味着需要服务器整体重启的部署工作也可以快速完成。此外，在 Unicorn 中，针对缩短部署时间还进行了其他一些优化。当 Unicorn-Master 进程收到 USR2 信号（稍后详述）时，Master 会进行下述操作步骤：

(1) 启动新 Master

Master 在收到 USR2 信号后，会启动 Ruby 解释器，运行一个新 Master 进程。

(2) 重新加载

新 Master 载入框架和应用程序代码。这个过程和 Thin 的重启一样，需要消耗一定的时间。但在这个过程中，旧 Master 依然在工作，因此没有任何问题。

① Copy-on-Write：是指在创建子进程时不复制父进程的内存空间，而是当内存数据发生写入时再进行复制的机制。（原书注）

(3) 启动 Slave

新 Master 通过 fork 系统调用启动 Slave，这样一来一个新版本的 Web 应用就准备完毕，可以提供服务了。

当新 Master 启动第一个 Slave 时，该 Slave 如果检测到存在旧 Master，则对其进程发送"QUIT"信号，命令旧 Master 结束进程。

然后，新 Master 开始运行新版本的应用程序。此时，旧 Master 依然存在，但服务的切换工作本身已经完成了。

(4) 旧 Master 结束

收到 QUIT 信号的旧 Master 会停止接受请求，并对 Slave 发出停止命令。旧 Slave 继续处理现存的请求，并在处理完毕后结束运行。当确认所有 Slave 结束后，旧 Master 本身也结束运行。到此为止，Unicorn 整体重启过程就完成了，而服务停止的时间为零。

6. 信号

在 Unicorn 重启的部分我们提到了"信号"这个概念。对于 UNIX 系操作系统不太了解的读者可能看不明白，信号也是 UNIX 中进程间通信的手段之一，但信号只是用于传达某种事件发生的通知而已，并不能随附其他数据。

信号的种类如表 2 所示，用 kill 命令可以发送给进程。

```
$ kill -QUIT <进程ID> ⏎
```

当 kill 命令中没有指定信号名时，则默认发送 INT 信号。在程序中则可以使用 kill 系统调用来发送信号，Ruby 中也有用于调用 kill 系统调用的 Process.kill 方法。

这些信号根据各自的目的都设有默认的动作，默认动作根据不同的信号分为 Term（进程结束）、Ign（忽略信号）、Core（内核转储）、Stop（进程暂停）、Cont（进程恢复）5 种。如果程序对于信号配置了专用的处理过程

表2　UNIX信号一览表(具有代表性的)

名　称	功　　能	默认动作
HUP	挂起	Term
INT	键盘中断	Term
QUIT	键盘终止	Core
ILL	非法命令	Core
ABRT	程序的 abort	Core
FPE	浮点数异常	Core
KILL	强制结束（不可捕获）	Term
SEGV	非法内存访问	Core
BUS	总线错误	Core
PIPE	向另一端无连接的管道写入数据	Term
ALRM	计时器信号	Term
TERM	结束信号	Term
USR1	用户定义信号 1	Term
USR2	用户定义信号 2	Term
CHLD	子进程暂停或结束	Ign
STOP	进程暂停（不可捕获）	Stop
CONT	进程恢复	Cont
TSTP	来自 tty 的 stop	Stop
TTIN	后台 tty 输入	Stop
TTOU	后台 tty 输出	Stop

（handler），则可以对这些信号进行特殊处置。不过，KILL 信号和 STOP 信号是无法改变处理过程的，因此即便因某个软件中配置了特殊的处理过程而无法通过 TERM 信号来结束，也可以通过发送 KILL 信号来强制结束。

信号原本是为特定状况下对操作系统和进程进行通知而设计的。例如，在终端窗口中按下 Ctrl+C，则会对当前运行中的进程发送一个 SIGINT。

然而，信号不光可以用来发送系统通知，也经常用来从外部向进程发送命令。在这些信号中，已经为用户准备了像 USR1 和 USR2 这两种可自定义的信号。

Unicorn 中也充分运用了信号机制。刚才我们已经讲到过，向 Slave 发送 QUIT 信号可以使其结束。Master/Slave 对于各个信号的响应方式如表 3 所示，其中有一些信号的功能看起来被修改得面目全非（比如 TTIN），这也算是“UNIX 流派”所特有的风格吧。

表3　Unicorn的信号响应

Master	
信　　号	动　　作
HUP	重新读取配置文件，重新载入程序
INT/TERM	立刻停止所有 Slave
QUIT	向所有 Slave 发送 QUIT 信号，等待请求处理完毕后结束
USR1	重新打开日志
USR2	系统重启。重启完成后当前 Master 会收到 QUIT 信号
TTIN	增加一个 Slave
TTOU	减少一个 Slave
Slave	
信　　号	动　　作
INT/TERM	立即停止
QUIT	当前请求处理完毕后结束
USR1	重新打开日志

信号还可以通过 Shell 或者其他程序来发送，因此，编写一个用于从外部重启 Unicorn 的脚本也是很容易的。

性能

在 http://github.com/blog/517-unicorn 专栏中，对 Unicorn 的性能与 Mongrel 进行了比较。根据这篇评测，无调优默认状态下的 Unicorn，性能已经稍优于 Mongrel 了。尽管 Thin 比 Mongrel

的性能更好一些，但 Unicorn 决不会甘拜下风的。

考虑到 Unicorn 几乎全部是用 Ruby 编写的（除 HTTP 报头解析器外）这一点，可以说是实现了非常优秀的性能。此外，从刚才所介绍的 Unicorn 的特点来看，在大多数情况下，用 Unicorn 来替代 Mongrel 和 Thin 还是有一定好处的。

不过，Unicorn 也并非万能。Unicorn 只适合每个请求处理时间很短的应用，而对于应用程序本身来说，在外部（如数据库查询等）消耗更多时间的情况，则不是很适合。

对于 Unicorn 来说，最棘手的莫过于像 Comet[①]这种，服务器端基本处于待机状态，根据状况变化推送响应的应用了。在 Unicorn 中，由于每个请求需要由一个进程来处理，这样会造成 Slave 数量不足，无法满足请求的处理，最终导致应用程序整体卡死。对于这样的应用程序，应该使用其他的一些技术，使得通过少量的资源就能够接受大量的连接。

为了弥补 Unicorn 的这些缺点，又出现了一个名叫“Rainbows!”[②]的项目。在 Rainbows! 中，可以对 N 个进程分配 M 个请求，从而缓和大量的连接数和有限的进程数之间的落差。

策略

综上所述，Unicorn 的关键是，不是由 HTTP 服务器来主动进行负载均衡，而是采用了完成工作的 Slave 主动获取请求的拉模式。对于 Slave 之间的任务分配则通过操作系统的任务切换来完成。这个案例表明，在大多数情况下，与其让身为用户应用的 HTTP 服务器来进行拙劣的任务分配，还不如将这种工作交给内核这个资源管理的第一负责人来完成。

另一个关键是对 UNIX 系操作系统功能的充分利用。例如，通过 fork 系统调用以及背后的 Copy-on-Write 技术加速 Slave 的启动。UNIX 中最近才加入了线程功能，Unicorn 则选择不依赖线程，而是对已经“过气”的进程技术加以最大限度的充分利用。线程由于可以共享内存空间，从性能上来说比进程要更加有利一些。但反过来说，正是因为内存空间的共享，使得它容易引发各种问题。因此 Unicorn 很干脆地放弃了使用线程的方法。

如此，Unicorn 充分利用了 UNIX 系操作系统长年积累下来的智慧，在保持简洁的同时，提供了充分的性能和易管理性。

① Comet：一种从 Web 服务器向 Web 客户端发送数据的推送（push）技术。（原书注）
② Rainbows! 项目官方网站：http://rainbows.rubyforge.org/。（原书注）

近年来，由于考虑到 C10K 问题（客户端超过一万台的问题）而采用事件驱动模型等，Web 应用程序也在用户应用程序的水平上变得越来越复杂。但 Unicorn 却通过将复杂的工作交给操作系统来完成，从而实现了简洁的架构。因为事件处理、任务切换等等本来就是操作系统所具备的功能。当然，仅凭 Unicorn 在客户端并发连接的处理上还是存在极限，如果请求数量过大也有可能处理不过来，但我们可以使用反向代理，将多个 Unicorn 系统捆绑起来，从而实现横向扩展（图 7）。

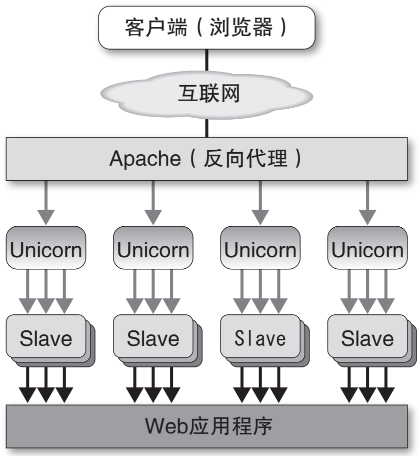


图 7　Unicorn 的横向扩展

小结

Unicorn 最大限度利用了 UNIX 的优点，同时实现了高性能和易管理性。此外，它采用了进程模式而非线程模式、拉模式而非推模式，通过追求实现的简洁，实现了优秀的特性，对于这一点我非常喜欢。今后，随着服务器端多任务处理的需求不断增加，像 Unicorn 这样简洁的方式会越来越体现其价值。

“云计算时代的编程”后记

首先，关于 HashFold 我想做一些补充。HashFold 从首次出现在我的专题连载中，到现在已经过了两年半的时间，它不但没有引起广泛关注，反倒是完全消亡了。对于使用 Hash 而非流（stream）的这个主意我觉得很有趣，但仅凭有趣还是无法推动潮流的吧。只不过，文章的内容本身，作为使用线程和进程来进行数据处理的实例来说，还是有足够的价值的，因此我还是决定将它放在这本书中。

现在反观 HashFold，在大量数据的处理上，比起运用 Hash 这样“容器”型数据结构的模型来说，我感觉“流”处理的方式在印象上要更好一些。此外，HashFold 真的要普及的话，最重要的是需要像 Hadoop 这样对 MapReduce 的高性能实现，而仅凭纸上谈兵恐怕是不会有什么结果的。

在思考今后“云计算时代的编程”这个话题的时候，本章中介绍的内容应该还会作为基础的技术继续存在下去，但程序员所看到的“表面”的抽象程度应该会越来越高。

今后，随着云计算的普及，节点的数量也会不断增加，对每个节点进行管理也几乎会变成一件不可能完成的事情。于是，节点就成了性能实现的一个“单位”，而作为一个单个硬件的节点概念则会逐步被忽略。在这样的环境中，恐怕不会再进行以显式指定节点的方式通信这样的程序设计了吧。说不定，在 Linda 这个系统中所提供的“黑板模型”会再次引起大家的关注。

这种模型是利用一块共享的黑板（称为 tuple space），先在上面写入信息，需要的人读取上面的信息并完成工作，再将结果写到黑板上。在 Ruby 中也利用 dRuby 提供了一个叫做 Rinda 的系统。

虽然 Linda 是 20 世纪 80 年代的一项古老的技术，但借着云计算的潮流，在这个领域中也不断要求我们温故而知新吧。

第 5 章

支撑大数据的数据存储技术

5.1 键 – 值存储

键 – 值存储（Key-value store）是数据库的一种。在云计算愈发流行的今天，键 – 值存储正在受到越来越多的关注。以关系型数据库管理系统（RDBMS）为代表的现有数据库系统正接近其极限，而键 – 值存储则拥有超越这种极限的可能性。

键 – 值存储是通过由键对象到值对象的映像来保存数据的，具体原理我们稍后会详细讲解。

例如，旅游预订网站“乐天旅游”[1]中可以显示“最近浏览过的酒店”，其数据中，键为“用户 ID”，值为“酒店 ID（多个）”，即通过和用户 ID 相关联，来保存用户浏览过的酒店 ID。

对于熟悉 Ruby 的读者，可以将这种方式理解为和 Ruby 内建的 Hash 类具有相同的功能。但不同的是，Hash 只能存在于内存中，而键 – 值存储是数据库，因此它具备将数据永久保存下来的能力。

使用键 – 值存储方式的数据库，大多数都在数据查找技术上使用了散列表这种数据结构。散列表是通过调用散列函数来生成由键到散列值（一个和原始数据一一对应的固定位数的数值）的映射，通过散列值来确定数据的存放位置。散列表中的数据量无论如何增大，其查找数据所需的时间几乎是固定不变的，因此是一种非常适合大规模数据的技术。

要讲解键 – 值存储，我们先从它的基本工作方式 Hash 开始讲起吧。

Hash 类

一般意义上说，Hash（散列表）指的是通过创建键和值的配对，由键快速找到值的一种数据结构。

作为例子，我们来看一看 Ruby 的 Hash 类。该类拥有 147 种方法，不过其本质可以通过下列 3 个方法来描述。

① 乐天旅游（楽天トラベル）：http://travel.rakuten.co.jp/

```
hash[key]
hash[key] = value
```

```
hash.each {|k,v| ...}
```

hash[key] 方法用于从 Hash 中取出并返回与 key 对象相对应的 value 对象。当找不到与 key 相对应的对象时，则返回 nil。hash[key] = value 方法用于将与 key 对象相对应的 value 对象存放到 Hash 中。当已经存在与 key 相对应的对象时，则用 value 覆盖它。最后是 hash.each 方法，用于按顺序遍历 Hash 中的键 - 值对。

也就是说，Hash 对象是用于保存 key 对象到 value 对象之间对应关系的数据结构。这种数据结构在其他编程语言中有时也被称为 Map（映像）或者 Dictionary（字典）。我觉得用字典这个概念来描述 Hash 的性质挺合适的，因为字典就是从一个词条查询其对应释义的工具。

DBM 类

Hash 类中的数据只能存在于内存中，在程序运行结束之后就会消失。为了超越进程的范围保存数据，可以使用 Ruby 的“DBM”类这样的键 - 值存储方式。

DBM 类的用法和 Hash 几乎一模一样，但也有以下这些区别：

- key 和 value 只能使用字符串。
- 创建新 DBM 对象时，需要指定用于存放数据的文件路径名称。
- 数据会被保存在文件中。

像这样，可以超越进程的范围来保存数据的特性，在编程的世界中被称为“永久性”（persistence）。

数据库的 ACID 特性

下面我们来分析一下“为什么在云计算时代键 - 值存储模型会受到关注”。

问题的关键在于 RDBMS 数据库所具备的 ACID 这一性质，我们就从这里开始讲起。ACID 是 4 个单词首字母的缩写，它们分别是：Atomicity（原子性）、Consistency（一致性）、Isolation（隔离性）和 Durability（持久性）。

所谓 Atomicity，是指对于数据的操作只允许“全部完成”或“完全未做改变”这两种状态

中的一种，而不允许任何中间状态。因为操作无法进一步进行分割，所以用了“原子”这个词来表现。例如，银行在进行汇款操作的时候，要从 A 账户向 B 账户汇款 1 万元，假设当中由于某些原因发生中断，这时 A 账户已经扣掉 1 万元，而 B 账户中还没有存入这 1 万元，这就是一个中间状态。

“从 A 账户余额中扣掉 1 万元”和“向 B 账户余额中增加 1 万元”这两个操作，如果只完成了其中一个的话，两个账户的余额就会发生矛盾。

所谓 Consistency，是指数据库的状态必须永远满足给定的条件这一性质。例如，当给定“存款账户余额永远为正数”这一条件时，“取出大于账户余额的款项”这一操作就无法被执行。

所谓 Isolation，是指保持原子性的一系列操作的中间状态，不能由其他事务进行干涉这一性质，由此可以保持隔离性而避免对其他事务产生影响。

所谓 Durability，是指当保持原子性的一系列操作完成时，其结果会被保存并且不会丢失这一性质。

整体来看，ACID 非常重视数据的完整性，而 RDBMS 正是保持着这样的 ACID 特性而不断进化至今的。

但近年来，要满足这样的 ACID 特性却变得越来越困难。这正是 RDBMS 的极限，也就是我们希望通过键 – 值存储来克服的问题。

CAP 原理

近年来，人类可以获得的信息量持续增加，如此大量的数据无法存放在单独一块硬盘上，也无法由单独一台计算机进行处理，因此通过多台计算机的集合进行处理成为了必然的趋势。这样一来，在实际运营时就会发生延迟、故障等问题。多台计算机之间的通信需要通过网络，而网络一旦饱和就会产生延迟。

计算机的台数越多，机器发生故障的概率也随之升高。在万台数量级的数据中心中，据说每天都会有几台计算机发生故障。当由于延迟、故障等原因导致“计算机的集合”之间的连接被切断，原本的集合就会分裂成若干个小的集合。

当组成计算环境的计算机数量达到几百台以上（有些情况下甚至会达到万台规模）时，ACID 特性就很难满足，换句话说，ACID 是不可扩展的。

对此，有人提出了 CAP 原理，即在大规模环境中：

- ❑ Consistency（一致性）
- ❑ Availability（可用性）
- ❑ Partition Tolerance（分裂容忍性）

这三个性质中，只能同时满足其中的两个。

在大规模数据库中如何保持 CAP，很难从一般系统的情况进行类推。因为在大规模系统中，延迟、故障、分裂都是家常便饭。

在我们平常所接触的数台规模的网络环境中，计算机的故障是很少发生的，但对于数万、数十万台规模的集群来说，这样的“常识”是无效的。在这样的数量级上，就会像墨菲定律所说的一样，“只要存在故障的可能性就一定会发生故障（而且是在最坏的时间点上）”。

据说 CAP 原理已经通过数学方法得到了证明。CAP 中的 C 是满足 ACID 的最重要因素，如果 CAP 原理真的成立的话，我们就可以推断，像 RDBMS 这样传统型数据库，在大规模环境中无法达到期望值（或者说无法充分发挥其性能）。这可真是个难题。

CAP 解决方案——BASE

根据 CAP 原理，C（一致性）、A（可用性）和 P（分裂容忍性）这三者之中，必须要舍弃一个。

如果舍弃分裂容忍性的话，那么只有两个选择：要么根本不会发生分裂，要么在发生分裂时能够令其中一方失效。

根本不会发生分裂，也就意味着需要一台能够处理大规模数据的高性能计算机。而且，如果这台计算机发生故障，则意味着整个系统将停止运行。现代的数据规模靠一台计算机来处理是不可能完成的，因此从可扩展性的角度来看，这并不是一个有效的方案。

此外，当发生分裂时，例如大的计算机集群被分割为两个小的集群，要区分哪一个才是“真身”也并非易事。如果准备一台主控机，以主控机所在的集群为“真身”，这的确可以做到。但如果主控机发生故障的话，就等于整个系统发生了故障，风险也就大大增加了。

在分布式系统中，像这样“局部故障会导致整体故障”的要害，被称为 Single point of failure（单一故障点），在分布式系统中是需要极力避免的。

不能舍弃可用性

那么，舍弃 A（可用性）这个选择又如何呢？这里的关键字是“等待”。也就是说，当发生分裂时，服务需要停止并等待分裂的恢复。另外，为了保持一致性，也必须等待所有数据都确实完成了记录。

然而，用户到底能够等待多长时间呢？仅仅作为一个用户的我，是相当没有耐心的，等上几秒钟就开始感到烦躁了，如果几分钟都没有响应，我想我就再也不会使用这个服务了。假设分裂和延迟的原因是由于机器故障，即便是准备了完善的备份机制，想要在几秒钟之内恢复也几乎是不可能的。所以结论就是，除非是不怎么用得上的服务，否则是不能舍弃可用性的。

那么现在就只剩下 C（一致性）了，舍弃一致性是否现实呢？仔细想想的话，在现实世界中严密的一致性几乎是不存在的。例如，A 要送个包裹给 B，在现实世界中是不可能瞬间送到的。A 需要将包裹交给物流公司，然后通过卡车等途径再送到 B 的手上，这个过程需要消耗一定的时间（无 Atomicity）。而且，配送中的状态是可以追踪的（无 Isolation），运输过程中如果发生事故包裹也可能会损坏（无 Consistency）。再有，即便对损坏和遗失上了保险，此次运输交易行为本身也不可能“一笔勾销”。

即便现实世界如此残酷，我们却还是进行着各种交易（事务）。这样看来，即便是在某种程度上无法满足一致性的环境中，数据处理也是能够完成的。例如，网上商城的商品信息页面上明明写着“有货”，到实际提交订单的时候却变成了“缺货”，这种事已经是家常便饭了，倒也不会产生什么大问题。和银行汇款不同，其实大多数处理都不需要严格遵循 ACID 特性。

在这样的环境中，BASE 这样的思路也许会更加合适。BASE 是下列英文的缩写：

- ❑ Basically Available
- ❑ Soft-state
- ❑ Eventually consistent

ACID 无论在任何情况下都要保持严格的一致性，是一种比较悲观的模式。而实际上数据不一致并不会经常发生，因此 BASE 比较重视可用性（Basically Available），但不追求状态的严密性（Soft-state），且不管过程中的情况如何，只要最终能够达成一致即可（Eventually consistent）。这种比较乐观的模式，也许更适合大规模系统。说句题外话，一开始我觉得 BASE 这个缩写似乎有点牵强，但其实 BASE（碱）是和 ACID（酸）相对的，这里面包含了一个文字游戏。

大规模环境下的键－值存储

好，下面该进入正题——键－值存储了。键－值存储的一个优点，是可以通过“给定键返回对应的值”这一简单的模式，来支撑相当大规模的数据。键－值存储之所以适合大规模数据，除了使用散列值这一点外，还因为它结构简单，容易将数据分布存储在多台计算机上。

不过，在实际的大规模键－值存储系统中，还是存在一些必须要注意的问题。下面我们通过一个大体的框架，来探讨一下可扩展的键－值存储架构。

分布键－值存储的基本架构并不复杂。多台计算机（节点）组成一个虚拟的圆环，其中每个节点负责某个范围的散列值所对应的数据。

当应用程序（客户端程序）对数据进行访问时，首先通过作为键的数据（字符串）计算出散列值，然后找到负责该散列值的节点，直接向该节点请求取出或者存放数据即可（图1）。怎么样，很简单吧？然而，要实现一个具备实用性和可扩展性的键－值存储系统，需要注意的问题还有很多。下面我们来看看这个系统的具体实现。

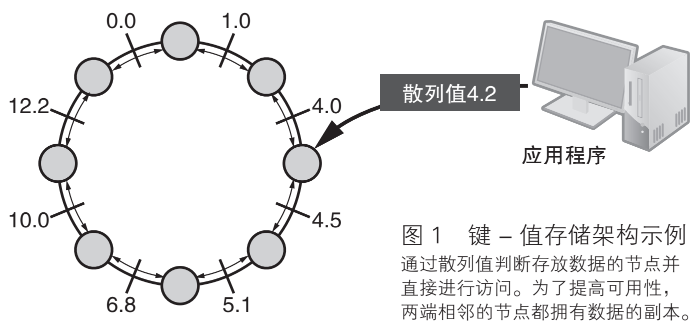


图1　键－值存储架构示例
通过散列值判断存放数据的节点并直接进行访问。为了提高可用性，两端相邻的节点都拥有数据的副本。

先声明一下，这里要讲解的可扩展键－值存储架构，基本上是以乐天开发的 ROMA 为基础的。可扩展键－值存储系统的实现有很多种，这里所介绍的架构并不是唯一一种。另外，为了讲解上方便，这里介绍的内容和实际的 ROMA 实现是有一些偏差的。

访问键－值存储

我们先来看一个简单的应用程序实现。应用程序一侧要执行的处理并不多，大体上可以分成“初始化”和“访问（获取、保存等）”两个步骤。

首先是初始化。应用程序在初始化时，需要指定几个构成键－值存储系统的节点。指定的节点并不需要是特殊节点，只要是参加键－值存储系统组成的节点都可以。之所以要指定多个，是考虑到在这其中至少应该有一个是存活的。

应用程序按顺序访问指定的节点，并向第一个应答的节点传达要访问键－值存储的请求。接着，建立连接的节点向客户端发送“哪个节点负责哪个范围的散列值”的信息（即路由表）。收到路由表之后，剩下的访问操作就比较简单了。根据要获取的键数据计算出散列值，然后通过路由表查询出负责该散列值的节点，并向该节点发送请求。请求的内容分为获取、保存等很多种类，在 ROMA 中所支持的请求如表 1 所示。比 Hash 要稍微复杂一些呢。

表1　ROMA的访问请求

访问请求	内　容
get	获取键所对应的值
set	设置键所对应的值
add	当键不存在时设置值
replace	当键存在时设置值
append	在当前值的末尾附加
prepend	在当前值的开头附加
delete	删除键所对应的值
inc	将键所对应的值加 1
del	删除键所对应的值

每次进行数据访问时，应用程序都需要与负责各个键（的散列值）的节点建立连接并进行通信。这个通信过程都是通过套接字来完成的。通过套接字与远程主机建立连接，实际上需要很大的开销。ROMA 早期的原型中，每次都需要建立这样的连接，于是这个部分就成了瓶颈，导致系统无法发挥出期望的性能。

所幸，一般情况下，对键－值存储的访问都具有局部性，也就是说对同一个键的访问可能会连续发生。在这样的情况下，池（pooling）技术就会比较有效。所谓池，就是指对使用过的资源进行反复利用的技术。这个案例中，也就是指对一定数量的套接字连接进行反复利用。特别是在访问具有局部性的情况下，连接池的效果是非常好的。

在键－值存储的运用中，难免会遇到由于延迟、故障、分裂等导致某些节点无法访问的状况。在 ROMA 中,各应用程序都持有一张记载组成键－值存储系统所有节点信息的表（路由表），并直接对节点进行访问。在这种类型的系统中，保持路由表处于最新状态是非常重要的。

ROMA 会定期对路由表进行更新。每隔一段时间，客户端会向路由表中的任意节点发出获取最新路由信息的请求。

此外，对各个节点的请求也设置了超时时间，如果某个节点未在规定时间内响应请求，则会被从路由表中删除。

键－值存储的节点处理

与应用程序相比，组成系统的节点的行为十分复杂，特别是像 ROMA 这样不存在承担特殊工作的主节点，且各节点之间相互平等的 P2P 型系统。

节点的工作大体包括以下内容：

- ❑ 应对访问请求
- ❑ 信息保存
- ❑ 维护节点构成信息
- ❑ 更新节点构成信息
- ❑ 加入处理
- ❑ 终止处理

正如图 1 所示，系统中的节点构成了一个圆环，其中每个节点的结构如图 2 所示。

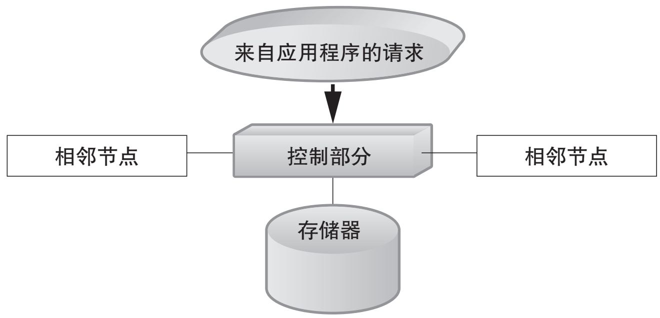


图 2　节点的结构

存储器

存储器（storage）就是实际负责保存信息的部分。在 ROMA 中，存储器是作为插件存在的，通过启动时的设置可以对存储器进行切换。目前实现的存储器包括下列这些：

- ❑ RH 存储器：将信息保存在 Ruby 的 Hash 对象中的存储器。这种方式无法将信息保存到文件中，因此 ROMA 整体运行停止后信息就消失了。
- ❑ DBM 存储器：将信息保存在 DBM（实际上是 GDBM）中的存储器。
- ❑ File 存储器：将信息保存在文件中的存储器。每个键都会产生一个独立的文件，因此磁盘目录可以用作索引。
- ❑ SQLite3 存储器：将信息保存在 SQLite3 中的存储器。SQLite3 是一种很有名的公有领域 RDBMS。
- ❑ TC 存储器：将信息保存在 Mixi 开发的 TokyoCabinet 中的存储器。是在 ROMA 实际运用中最为常用的一种。

各个存储器都定义为“Roma::Storage::BasicStorage”的子类，其定义采用模板方法的形式。在运用中，ROMA 可以根据数据库所需特性来选择合适的存储器。从实际来看，大多数案例都可以用 TC 存储器来解决。

写入和读取

当应用程序发起写入请求时，在确认键的散列值属于本节点负责范围后，该节点会通过存储器对数据进行保存。

这个时候，如果数据只保存在一个节点上的话，万一这个节点发生故障，数据就会丢失。为了提高可用性和分裂容忍性，写入必须由多个节点共同完成。

如果重视响应速度的话，可以在本节点完成写入操作之后，马上对请求进行响应，剩下的写入操作则可以在后台完成。在 ROMA 中由于对响应速度并不是非常重视，而是需要追求可靠性，因此所有的写入操作是同步执行的。不过，如果由于某些原因导致数据复制写入失败，则会在后台重新尝试执行写入操作。

如果由于应用程序所持有的路由表信息过期等原因，导致请求写入的键不属于本节点负责的范围，该节点会将请求转发出去。

读取的处理和写入差不多，但由于不需要出于冗余的目的对其他节点发出请求，因此处理方式更加简单。

节点追加

分布式键－值存储系统的优点就是运用的灵活性。当数据过多、访问速度下降时，只要追加新的节点就可以进行应对。

追加新节点时，需要指定一个现有的节点作为入口，然后启动新的节点。新启动的节点和指定的现有节点进行通信，申请加入环状结构，然后向全体节点发送对节点间数据分配进行调整的请求。刚刚启动的新节点并不包含任何数据（在启动时显式指定了已永久保存的数据库的情况除外），随着节点调整的进行，数据会逐步分配给新的节点。

故障应对

作为可扩展的键－值存储系统来说，最重要的恐怕就是对故障的容忍性了。正如之前讲过的，随着组成系统的计算机台数的增加，发生故障的概率也会大幅度上升。在大规模系统中，即便组成系统的一部分计算机发生故障，系统也必须能够继续运行。

发生频率最高的，应该是单台计算机的故障。由于故障导致一台计算机从系统中消失，这样的例子十分常见。

当由于故障导致一个节点失去响应时，应用程序会尝试访问其他节点。在 ROMA 中，由于数据总是存放在多个节点中，因此通过路由表就可以找到其他的替代节点。

另一方面，出现无响应的节点，就意味着数据的冗余度下降了。为了避免这种情况，其他节点需要将消失的节点排除，然后重新组织节点的结构，根据需要向相邻节点复制数据，最终维持数据的平衡。新的节点结构信息，会通过定期更新发送给应用程序，而作为整个键 - 值存储来说，会像什么都没有发生一般继续运行。

比较麻烦的情况是，暂时“消失”的节点又复活了。这种情况的发生可能是由于网线被拔出（这是在运营工作中经常会出现的意外），或者由于网络故障导致大规模网络延迟，这些应该还是比较常见的。

在以简洁为信条的 ROMA 中，遇到这样的情况，会将已经分离的节点完全舍弃。如果出现“复活”的情况，该节点需要作为一个新节点重新加入 ROMA 系统。ROMA 会将新加入的节点更新到路由表中，并重新对数据进行分配。

另一种故障也可能发生，那就是多个节点同时消失。在 ROMA 中，和一个节点消失的情况一样，会将这些节点舍弃。在多个节点同时消失的情况下，可能会发生冗余备份的数据同时丢失的问题。要找回丢失的数据是不可能的，因此系统就会报错。在这种情况下，就无法区分该数据是一开始就不存在，还是由于大量节点消失而导致的数据丢失。这当然会引发一些问题，但失去的东西总归无法复得，也就没必要进行任何特殊的处理了。

话虽如此，但丢失数据这种事，作为数据库来说确实是个不小的问题。为了拯救数据，ROMA 中提供了一个命令，可以将切断前已经由存储器写入文件的数据重新上传回 ROMA。这个操作只是将存储器数据读取出来并添加到 ROMA 中，基本的部分是非常简单的。不过，如果上传的数据中有一些键已经在 ROMA 中存在，且它们所对应的值不相同的情况下，必须决定选用其中某一个值来解决冲突。到底是分离之后 ROMA 一侧的数据被更新了，还是对节点的数据写入由于某些原因没有反映到 ROMA 一侧？仅凭键和值的数据是无法判断的。

实际上，在 ROMA 中对每份数据都附加了一个叫做“逻辑时钟”（logical clock）的信息，它是一种在每次数据被更新时进行累进的计数器。当上传的数据和 ROMA 中已经存在的数据发生冲突时，通过逻辑时钟就可以判断应该以哪一方的数据为准。

在各种故障中，还可能发生分裂的情况，也就是在完全隔绝的网络中，还在继续独立工作的意思。在 ROMA 中，要应对这种故障，只能将分裂开的 ROMA 系统中的其中一个手动停止。

虽然这种手段非常原始，但分裂这样的故障并不会经常发生，这样的应对应该已经足够了。从分裂故障中进行恢复，和上述情况一样，也是通过使用从存储器上传数据的功能来完成的。

终止处理

出人意料的是，在 P2P 型结构中，最麻烦的操作居然是终止。要进行终止操作，首先要向任意节点发送终止请求，然后，该节点就自动成为负责终止的主节点，由它对全体节点发送“即将终止”的声明。收到声明之后，各节点停止接受新的请求，并在当前时间点正在处理的请求全部完成之后，对存储器执行文件的写入（内存存储的情况除外），完成后向终止主节点发送回复。回复完毕后，结束该节点的进程。

负责终止的主节点在收到全部节点（故障、无应答的节点除外）的回复后，结束自身进程。至此，ROMA 系统的运行就全部停止了。

其他机制

除了上述讲到的内容之外，ROMA 中还有以下这些机制：

1. 有效期

ROMA 中的数据都设置了有效期，因此如果要实现“这个数据仅在今天有效，明天就需要删掉”这样的规则是很容易的。

2. 虚拟节点

为了让节点之间因分配调整而进行的数据传输更加高效，系统中采用了将若干个键组织起来形成“虚拟节点”的机制。

3. 散列树

如图 1 所示，ROMA 使用了环状节点分布和浮点小数散列值，实际上的算法使用的是 SHA-1[①]散列和 Merkle 散列树[②]，这种方式的效率更高。

---

① SHA-1 是 SHA 系列的成员之一。SHA（Secure Hash Algorithm，安全散列算法）是由美国国家安全局设计的一套散列算法，是美国的国家标准，被广泛应用于各种安全协议中。SHA 系列包括 SHA-1、SHA-224、SHA-256、SHA-384 和 SHA-512 共 5 个成员。

② 散列树（hash tree）是用于存放大量数据（如文件）的散列信息的一种数据结构，用于对数据进行验证。这种数据结构是列表和链表的组合，是对散列算法的一种扩展。散列树又被称为 Merkle 树，该命名来源于 Ralph Merkle（1952—　），他是公钥加密算法的创始人之一，同时也是一位研究人体冷冻技术和分子纳米技术的科学家。

性能与应用实例

在乐天旅游的“最近浏览过的酒店”和乐天市场[1]的“浏览历史”等功能中，都采用了ROMA，每个用户的访问历史记录都是保存在 ROMA 中的。ROMA 基本上都是用 Ruby 编写的，但是它所提供的性能却足够支持日本最大级网站的应用。

小结

除了 ROMA 之外，还有很多键－值存储系统的实现方式，它们也都具备各自的特点。由于这些项目大多数都是开源的，因此通过阅读源代码来研究一下或许也是一件很有意思的事。

① 乐天市场：http://www.rakuten.co.jp/

5.2 NoSQL

说起 NoSQL，这里并不是指某种数据库软件叫这个名字[①]。所谓 NoSQL，是一个与象征关系型数据库的 SQL 语言相对立而出现的名词，它是包括键 – 值存储在内的所有非关系型数据库的统称。不过，关系型数据库在很多情况下还是非常有效的，因此有人批判 NoSQL 这个词中所体现出的“不再需要 SQL”这个印象过于强烈，主张应该将其解释为“Not Only SQL”（不仅是 SQL）（图 1）。

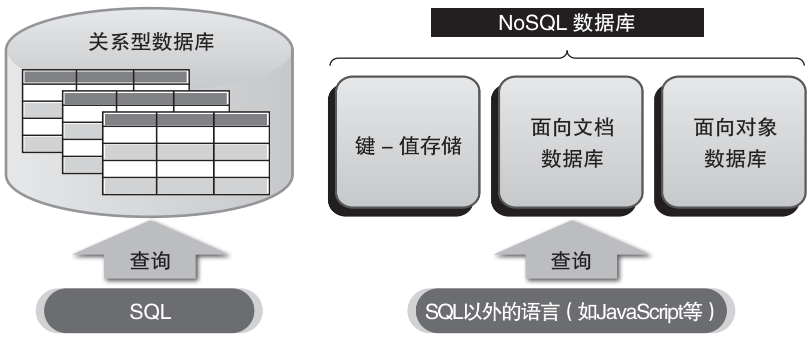


图 1　NoSQL 数据库

属于 NoSQL 类的数据库，主要有 ROMA（Rakuten On-Memory Architecture）这样的键 - 值存储型数据库，以及接下来要介绍的 MongoDB 这样的面向文档数据库等。

RDB 的极限

在大规模环境中，尤其是作为大流量网站的后台，一般认为关系型数据库在性能上存在极限，因为关系型数据库必须遵守 ACID 特性。

ACID 是 Atomicity（原子性）、Consistency（一致性）、Isolation（隔离性）和 Durability（持久性）这四个单词首字母的缩写。

① 我查了一下，真有一种数据库软件叫 NoSQL，不过这次我们的话题并不是指这个软件。（原书注）

所谓 Atomicity，是指对于数据的操作只允许“全部完成”或“完全未做改变”这两种状态中的一种，而不允许任何中间状态。因为操作无法进一步进行分割，所以用了“原子”这个词来表现。

所谓 Consistency，是指数据库的状态必须永远满足给定的条件这一性质。当某个事务无法满足给定条件时，其执行就会被取消。

所谓 Isolation，是指保持原子性的一系列操作的中间状态，不能由其他事务进行干涉这一性质，由此可以保持隔离性而避免对其他事务产生影响。

所谓 Durability，是指当保持原子性的一系列操作完成时，其结果会被保存并且不会丢失这一性质。

当数据量和访问频率增加时，ACID 特性就成了导致性能下降的原因，因为随着数据量和访问频率的增加，维持 ACID 特性所带来的开销就会越来越明显。

例如，为了保持数据的一致性，就需要对访问进行并发控制，这样则必然会导致能接受的并发访问数量下降。如果将数据库分布到多台服务器上，则为了保持一致性所带来的通信开销也会导致性能下降。

当然，如果以适当的方式将数据库分割开来，从而在控制访问频率和数据量方面进行优化的话，在一定程度上可以应对这个问题。在大规模环境下使用关系型数据库，一般有水平分割和垂直分割两种分割方式。

所谓水平分割，就是将一张表中的各行数据直接分割到多个表中。例如，对于像 mixi[①]这样的社交化媒体（SNS）网站，如果将用户编号为奇数的用户信息和编号为偶数的用户信息分别放在两张表中，应该会比较有效。

相对地，所谓垂直分割就是将一张表中的某些字段（列）分离到其他的表中。用 SNS 网站举例的话，相当于按照“日记”、“社区”等功能来对数据库进行分割。

通过这样的分割，可以对单独一个关系型数据库的访问量和数据量进行控制。但是这样做，维护的难度也随之增加。

NoSQL 数据库的解决方案

NoSQL 之所以受到关注，就是因为它可以成为解决关系型数据库极限问题的一种方案。和关系型数据库相比，NoSQL 数据库具有以下优势（图 2）：

① mixi：http://mixi.jp/，是日本最大的社交网站。

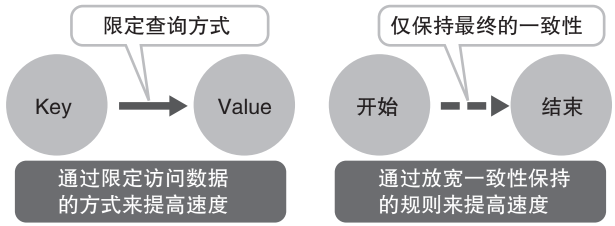


图 2　NoSQL 的优点

❑ 限定访问数据的方式

在大多数 NoSQL 数据库中，对数据访问的方式都被限定为通过键（查询条件）来查询相对应的值（查询对象数据）这一种。由于存在这样的限定，就可以实现高速查询。而且，大多数 NoSQL 数据库都可以以键为单位来进行自动水平分割。

此外，也有像 memcached 这样不永久保存数据，只是作为缓存来使用的数据库。这也算是一种对数据访问方式的限定吧。

❑ 放宽一致性原则

要保持大规模数据库，尤其是分布式数据库的一致性，所需要的开销十分显著。因此大多数 NoSQL 数据库都遵循“BASE”这一原则。

所谓 BASE，是 Basically Available、Soft-state 和 Eventually consistent 的缩写，ACID 无论在任何情况下都要保持严格的一致性，而实际上数据不一致并不会经常发生，因此 BASE 比较重视可用性（Basically Available），但不追求状态的严密性（Soft-state），且不管过程中的情况如何，只要最终能够达成一致即可（Eventually consistent）。

如果遵循 BASE 原则，那么用于保持一致性的开销就可以得到控制，而标榜 ACID 的关系型数据库则很难做出这样的决断。

形形色色的 NoSQL 数据库

NoSQL 数据库只是一个统称，其中包含各种各样的数据库系统。大体上，可以分为以下三种：

❑ 键 – 值存储数据库
❑ 面向文档数据库
❑ 面向对象数据库

键 – 值存储是一种让键和值进行关联的简单数据库，查询方式基本上限定为通过键来进行，

可以理解为在关系型数据库中只能提供对“拥有特定值的记录”进行查询的功能，而且还是有限制的。在UNIX中从很早就提供的DBM这种简单数据库，从分类上来看也可以算作键-值存储，但是在 NoSQL 这个语境中，所谓键－值存储一般都指的是分布式键－值存储系统。符合这样条件的键－值存储数据库包括“memcached”、“ROMA”、“Redis”、“TokyoTyrant”等。

所谓面向文档数据库，是指对于键－值存储中“值”的部分，存储的不是单纯的字符串或数字，而是拥有结构的文档。和单纯的键－值存储不同，由于它可以保存文档结构，因此可以基于文档内容进行查询。

举个例子，一张会员清单包括姓名、地址和电话号码，现在要从中查找一个名字叫“松本”的会员。也许乍看之下这和关系型数据库的应用方式是一样的，但是不同之处在于，在面向文档数据库中，对于存放会员信息的文档来说，每个会员的文档结构可以是不同的。因此要查找名字叫“松本”的会员，实际上相当于对“具备名字这个属性，且该属性的值为松本的文档”进行查询。这种情况下的文档，通常采用的是 XML（eXtended Markup Language）和 JSON（JavaScript Object Notation）格式。面向文档数据库包括 CouchDB、MongoDB 以及各种 XML 数据库等。

所谓面向对象数据库，是将面向对象语言中的对象直接进行永久保存，也就是当计算机断电关机之后对象也不会消失的意思。键－值存储和面向文档数据库给人的感觉还像是个数据库，但大多数面向对象数据库看起来只是一个将对象进行永久保存的系统而已。当然，面向对象数据库也提供对对象的查询功能。面向对象数据库的例子有 Db4o、ZopeDB、ObjectStore 等。

在我跳槽到现在的公司之前，经常使用 ObjectStore。那时候的工作内容是用 C++ 和 ObjectStore 编写一个 CAD 软件，真是怀念啊。说起来，那个时候 ObjectStore 还不支持分布式环境，对于跨越多数据库创建对象的功能，以及对不再使用的对象进行回收的分布式垃圾回收功能等，都是靠自己的力量实现的，不知道现在是不是有了正式的支持呢。

从“非关系型数据库”的角度来看，在这里我们暂且将面向对象数据库也算作是 NoSQL 的一种，至少从我（有些过时）的经验来说，面向对象数据库的主要目的，是提升一些数据结构比较复杂的小规模数据库的访问速度，而和其他 NoSQL 数据库相比，在可扩展性方面并不是很擅长。

面向文档数据库

下面我们来介绍一下面向文档数据库。所谓面向文档数据库，可以理解为是将 JSON、XML 这样的文档直接进行数据库化的形式，其特点包括：不需要 schema（数据库结构定义），

支持由多台计算机进行并行处理的“水平扩展”等。

1. CouchDB

CouchDB 可以说是面向文档数据库的先驱。CouchDB 的特点是 RESTful 接口以及采用 Erlang 进行实现。

CouchDB 提供了遵循 REST（Representational State Transfer，表征状态转移）模型的接口，因此，即便没有特殊的客户端和库，使用 HTTP 也可以对数据进行插入、查询、更新和删除操作。和关系型数据库不同，其中每条数据不必拥有相同的结构，可以各自拥有一些自由的元素。在 CouchDB 中，是通过 JSON 来对记录进行描述的。

此外，在 CouchDB 中，一部分逻辑可以用 JavaScript 编写并插入到数据库中，从整体上看，数据库和应用程序之间的区别并不是那么明确。大多数人都习惯于“数据库负责数据，应用程序负责逻辑”，但此时也许需要让自己从这种模式中跳出来。

出人意料的是，像数据表的连结（Join）之类，在传统数据库中通过 SQL 可以轻松完成的查询，在 CouchDB 中是做不到的。因此用惯了传统关系型数据库的人可能会觉得四处碰壁。但是，如果能够完全运用 CouchDB 的功能，应用程序的设计可以变得十分简洁。

这种数据库是用 Erlang 来实现的，这一点也很值得关注。Erlang 是一种为并行计算特别优化过的函数型语言，分布式计算和并行计算方面的程序设计一直是它的强项，因此在 CouchDB 这样需要通过多台机器的分布和协调应对大量访问的场景中，应该说能够充分发挥 Erlang 的性能。

2. MongoDB

和 CouchDB 相比，MongoDB 大概更接近传统的数据库。MongoDB 的宣传口号是 Combining the best features of document databases, key-value stores, and RDBMSes，即要结合（像 CouchDB 这样的）文档数据库、键－值存储数据库和关系型数据库的优点，这真是个颇具挑战的目标。

MongoDB 除了不具备事务功能之外，确实提供了和关系型数据库非常接近的易用性。此外，它还为 C++、C#、JavaScript、Java、各种 JVM 语言、Perl、PHP、Python、Ruby 等语言提供了访问驱动程序，这一点也非常重要。有了这样的支持，在语言的选择上也就没有什么顾虑了。

MongoDB 的安装

如果你所使用的操作系统发行版本中提供了 MongoDB 的软件包，那么安装就非常容易了。在 Debian 中该软件包的名字叫做 mongodb。

即便没有提供软件包，安装它也并非难事。只要访问 MongoDB 官方网站的下载页面：http://www.mongodb.org/downloads，找到对应的二进制包并下载就可以了。提供官方预编译版本的系统平台如表 1 所示。

我选用的是 Linux 32 位版本。将下载好的 tar.gz 文件解压缩后，其目录结构如下：

表1 MongoDB提供预编译版本的系统平台

Mac OS X 32 位
Mac OS X 64 位
Linux 32 位
Linux 64 位
Windows 32 位
Windows 64 位
Solaris x86
Solaris 64 位

```
GNU-AGPL-3.0（许可协议）
README（说明文件）
THIRD-PARTY-NOTICES（第三方依赖关系信息）
bin/（二进制文件）
include/（头文件）
lib/（库文件）
```

MongoDB 的许可协议是 GNU-AGPL-3.0。AGPL 这种协议可能大家没怎么听说过，它是 AFFERO GENERAL PUBLIC LICENSE 的缩写，简单讲，基本条款和 GPL 是差不多的，区别只有一点，就是在该软件是通过网络进行使用的情况下，也需要提供源代码。在用于商业用途的情况下，如果不想公开源代码，貌似也可以购买商用许可。

bin 目录中包含了 MongoDB 的数据库服务器、客户端、工具等可执行文件。只要将这些文件复制到 /usr/bin 等 Path 能搜索到的目录中就可以完成安装了。如果需要自行编译客户端和驱动程序的话，还需要安装 include 目录中的头文件和 lib 目录中的库文件。

如果没有和你所使用的操作系统或 CPU 相对应的预编译版本，则需要下载源代码自行编译。不过，MongoDB 所依赖的库有很多，准备起来也有点麻烦。如果要在 Ubuntu 下用源代码进行编译，可以参考这里的资料（英文）：http://www.mongodb.org/display/DOCS/Building+for+Linux。

接下来我们需要用 Ruby 来访问 MongoDB，因此还需要安装 Ruby 的驱动程序。用 RubyGems 就可以轻松完成安装。RubyGems 是为 Ruby 的各种库和应用程序设计的软件包管理系统，使用起来非常方便。如果你还没有安装 RubyGems 的话，趁这个机会赶紧安装吧。在 Debian 或 Ubuntu 中，输入下列命令进行安装：

```
$ sudo apt-get install ruby rubygems ↵
```

↵表示换行

用 RubyGems 来安装 MongoDB 的 Ruby 驱动程序，可以输入下列命令：

```
$ sudo gem install mongo ↵
```

启动数据库服务器

启动数据库服务器的命令是 mongod，作为参数需要指定数据库存放路径以及 mongod 监听连接的端口号，默认的端口号为 27017。指定数据库路径的选项为“--dbpath”，指定端口号的选项为“--port”。例如，如果创建一个“/var/db/mongo”目录并希望将数据库存放在此处，可以用下面的命令来启动数据库服务器（假设 mongod 所在的路径能够被 Path 找到，如果不能的话则需要指定绝对路径）：

```
$ sudo mongod --dbpath /var/db/mongo ⏎
```

服务正常启动后会显示“waiting for connections on port 27017”这样一条消息（屏幕截图 1）。



屏幕截图 1　MongoDB 启动时的样子

对 MongoDB 进行操作需要使用 mongo 命令。如果为数据库服务器指定了非默认的端口号，则 mongo 命令也需要指定 --port 参数。打开一个新的终端控制台，用下列命令来启动 mongo：

```
$ mongo ⏎
MongoDB shell version: 1.3.1
url: test
connecting to: test
type "exit" to exit
type "help" for help
>
```

这样就连接成功了。这个命令可以通过交互的方式对数据库进行操作，对于学习 MongoDB 很有帮助。此外，对于数据库的小规模调整和修改也十分方便。

不过 mongo 命令没有提供行编辑功能，如果配合使用支持行编辑功能的 rlwrap 命令则会更加方便。

```
$ rlwrap mongo ⏎
```

用上述格式启动，就可以为 mongo 命令增加行编辑功能。这样不仅能对输入行进行编辑，还可以查询输入历史，非常方便。

在 Debian 和 Ubuntu 中，可以用下列命令来安装 rlwrap：

```
$ sudo apt-get install rlwrap ⏎
```

MongoDB 的数据库结构

MongoDB 的结构分为数据库（database）、集合（collection）、文档（document）三层。在 mongo 命令中输入“show dbs”可以显示当前连接的数据库服务器所管理的数据库清单。

```
> show dbs ↵
admin
local
```

我们可以看到，这台服务器所管理的数据库有 admin 和 local 这两个。对数据库的操作是针对当前数据库进行的。在连接时显示的消息中，“connecting to:”所表示的就是当前数据库。查看当前数据库可以使用“db”命令：

```
> db ↵
test
```

在这里，数据库包含若干个集合，而集合则相当于关系型数据库中“表”的概念。关系型数据库中的表，都拥有各自的结构定义（schema），结构定义决定了表中各行（记录）包含怎样的数据，以及这些数据排列的顺序。因此每条记录都遵循 schema 的定义而具备完全相同的结构。

但对于无结构的 MongoDB 数据库来说，虽然集合中包含了相当于记录的文档，但每一个文档并不必具备相同的结构，而是能够存放可以用 JSON 进行描述的任意数据。一般来说，在同一个集合中倾向于保存结构相同的文档，但 MongoDB 对此并非强制。

这就意味着，随着应用程序开发的进行，对于数据库中数据的结构变化，可以灵活地做出应对。在 Ruby on Rails 的开发中，一旦数据库结构发生变化，就必须花很大精力来编写数据迁移脚本，而这样的苦差事在 MongoDB 中是完全可以避免的。

数据的插入和查询

在关系型数据库中，要创建新的表，需要对表结构进行明确的定义并执行显式的创建操作，而在更加灵活的 MongoDB 中则不需要这么麻烦。在 mongo 命令中用 use 命令可以切换当前数据库，如果 use 命令指定了一个不存在的数据库，则会自动创建一个新数据库。

```
> use linux_mag ↵
switched to db linux_mag
```

而且，如果向不存在的集合中保存文档的话，就会自动创建一个新的集合。

```
> db.articles.save({ ↵
... title: "技术的剖析", ↵
... author: "matz" ↵
... }) ↵
```

其中“…”是 mongo 命令中表示折行的提示符。通过这样的命令，我们就向 linux_mag 数据库的 articles 集合中插入了一个新文档。

```
> show collections ↵
articles
system.indexes
```

下面我们来查询一下这个文档。查询文档需要使用集合的 find 方法。

```
> db.articles.find() ↵
{ "_id" : ObjectId("4b960889e4ffd91673c93250"), "title" : "技术的剖析",
"author" : "matz" }
```

保存的数据会被自动分配一个名为“_id”的唯一 ID。find 方法还可以指定查询条件，如：

```
> db.articles.find({author: "matz"}) ↵
{ "_id" : ObjectId("4b960889e4ffd91673c93250"), "title" : "技术的剖析",
"author" : "matz" }
```

如果指定一个 JavaScript 对象作为 find 方法的参数，则会返回与其属性相匹配的文档。在这里我们的数据库中只有一个文档，如果有多个匹配的文档的话，自然会返回多个结果。

如果希望只返回一个符合条件的文档，则可以用 findOne 方法来代替 find 方法。

用 JavaScript 进行查询

mongo 命令最重要的一点，是可以自由地运行 JavaScript。mongo 所接受的命令，除了 help、exit 等一部分命令之外，其余的都是 JavaScript 语句。甚至可以说，mongo 命令本身就是一个交互式的 JavaScript 解释器。刚才的例子中出现的：

```
db.articles.find()
```

等写法，正是 JavaScript 的方法调用形式。由于支持 JavaScript，因此我们可以自由地进行一些简单的计算，将结果赋值给变量，甚至用 for 语句进行循环。

```
> 1 + 1 ↵
2
> print("hello") ↵
hello
```

下面我们就用 JavaScript 来为数据库填充一定规模的数据。

```
> for (var i = 0; i < 1000000
; i++) { ... db.bench.save( { x:4, j:i } ); ... }
```

在花了相当长一段时间之后，我们就创建了一个包含 100 万个文档的 bench 集合。接下来，我们来试试看查询。

```
> db.bench.findOne({j:999999})
{ "_id" : ObjectId("4b965ef5ffa07ec509bd338e"), "x" : 4, "j" : 999999 }
```

在我的电脑上查询到这个结果用了差不多 1 秒的时间。因为要将 100 万个文档全部查询一遍，所以这个速度不是很快，于是我们来创建一个索引。

```
> db.bench.ensureIndex({j:1}, {unique: true})
```

这样我们就对 j 这个成员创建了一个索引，再查询一次试试看。

```
> db.bench.findOne({j:999999})
{ "_id" : ObjectId("4b965ef5ffa07ec509bd338e"), "x" : 4, "j" : 999999 }
```

在创建索引之前，按下回车键到返回结果总觉得会卡一会儿，现在则是瞬间就可以得到结果，索引的效果是非常明显的。

在 mongo 命令中，可以使用 JavaScript 这样一种完全编程语言来对数据库进行操作，感觉真是不错。也许是因为我没有在工作中使用过 SQL 的原因吧，总觉得需要用 SQL 这样一种不完全语言来编写算法和进行操作的关系型数据库让我觉得不太习惯，相比之下还是 MongoDB 感觉更加亲近一些。从我个人的喜好来说，自然希望在 mongo 命令中也可以用 Ruby 来对数据库进行操作。话说，2012 年 4 月，AvocadoDB[①]宣布要集成 mruby，这很值得期待。

高级查询

在 MongoDB 中，使用 find 或者 findOne 方法并指定对象作为条件时，会返回成员名称和值相匹配的文档。严格来说，find 方法返回的是与符合条件的结果集相对应的游标，而 findOne 则是仅返回第一个找到的符合条件的文档。

不过，说到查询，可并不都是这么简单的用法。下面我们通过将 SQL 查询转换为 MongoDB 查询的方式，来学习一下 MongoDB 的查询编写方法。刚才出现过的查询符合条件记录的例子，用 SQL 来编写的话应该是下面这样：

① AvocadoDB 已于 2012 年 5 月改名为 ArangoDB: http://www.arangodb.org/

```
SELECT * FROM bench
 WHERE x = 4
```

这条查询写成 MongoDB 查询则是这样：

```
> db.bench.find({x: 4}) ⏎
```

如果希望只选出特定的成员（字段），用 SQL 要写成：

```
SELECT j FROM bench WHERE x = 4
```

MongoDB 的话则是：

```
> db.bench.find({x: 4}, {j: true}) ⏎
```

刚才我们的查询条件都是“等于”，如果要比较大小当然也是可以的。例如，“x 大于等于 4”这样的条件，用 SQL 查询可以写成：

```
SELECT j FROM bench WHERE x >= 4
```

而 MongoDB 的话则是：

```
> db.bench.find({x: 4}, {j: {$gte: 4}}) ⏎
```

当比较条件不是等于时，要像上面这样使用以“$”开头的比较操作符来表达。MongoDB 中可以使用的比较操作符如表 2 所示。

表2　比较操作符

名　　称	语　　法	含　　义
$gt	{$gt: val}	大于 val
$lt	{$gt: val}	小于 val
$gte	{$gte: val}	大于等于 val
$lte	{$lte: val}	小于等于 val
$ne	{$ne: val}	不等于 val
$in	{$in: val}	包含 val
$nin	{$nin: val}	不包含 val
$mod	{$mod: [n, m]}	除以 n 的余数为 m
$all	{$all: ary}	包含 ary 的所有元素
$size	{$size: n}	数组长度为 n
$exists	{$exists: true}	存在
$exists	{$exists: false}	不存在
$not	{$not: cond}	否定条件
正则表达式 /	^foo	等正则匹配
$where	{$where: str}	将 str 作为 JavaScript 进行求值，用 this 来引用文档

我们刚才已经讲过，在 find 方法中，返回的并不是文档本身，而是游标（cursor）。当执行的查询得到多个匹配结果时，某些情况下返回的结果数量可能会超乎想象。这时，我们可以使用 count、limit、skip、sort 等方法。

count 方法可以返回游标所关联的结果集的大小。

```
> db.bench.find().count() ↵
 1000000
```

limit 方法可以将结果集的大小限制为从游标开始位置起指定数量的文档（图 3）。

skip 方法可以使游标跳过指定数量的记录（图 4）。配合使用 limit 和 skip，就可以像 Google 搜索页面一样，轻松实现以 n 个结果为单位将结果进行分页的操作。

```
> db.bench.find().limit(10) ↵
 { "_id" : ..., "x" : 4, "j" : 0 }
 { "_id" : ..., "x" : 4, "j" : 1 }
 { "_id" : ..., "x" : 4, "j" : 2 }
 { "_id" : ..., "x" : 4, "j" : 3 }
 { "_id" : ..., "x" : 4, "j" : 4 }
 { "_id" : ..., "x" : 4, "j" : 5 }
 { "_id" : ..., "x" : 4, "j" : 6 }
 { "_id" : ..., "x" : 4, "j" : 7 }
 { "_id" : ..., "x" : 4, "j" : 8 }
 { "_id" : ..., "x" : 4, "j" : 9 }
```

图 3　limit 方法的执行结果

```
> db.bench.find().skip(10).limit(10)
 { "_id" : ..., "x" : 4, "j" : 10 }
 { "_id" : ..., "x" : 4, "j" : 11 }
 { "_id" : ..., "x" : 4, "j" : 12 }
 { "_id" : ..., "x" : 4, "j" : 13 }
 { "_id" : ..., "x" : 4, "j" : 14 }
 { "_id" : ..., "x" : 4, "j" : 15 }
 { "_id" : ..., "x" : 4, "j" : 16 }
 { "_id" : ..., "x" : 4, "j" : 17 }
 { "_id" : ..., "x" : 4, "j" : 18 }
 { "_id" : ..., "x" : 4, "j" : 19 }
```

图 4　skip 方法的执行结果

sort 方法可以按指定成员对查询结果进行排序（图 5）。

```
> var c = db.bench.find()
 > c.skip(10).limit(10).sort({j: -1})
 { "_id" : ..., "x" : 4, "j" : 999989 }
 { "_id" : ..., "x" : 4, "j" : 999988 }
 { "_id" : ..., "x" : 4, "j" : 999987 }
 { "_id" : ..., "x" : 4, "j" : 999986 }
 { "_id" : ..., "x" : 4, "j" : 999985 }
 { "_id" : ..., "x" : 4, "j" : 999984 }
 { "_id" : ..., "x" : 4, "j" : 999983 }
 { "_id" : ..., "x" : 4, "j" : 999982 }
 { "_id" : ..., "x" : 4, "j" : 999981 }
 { "_id" : ..., "x" : 4, "j" : 999980 }
```

图 5　sort 方法的执行结果

这样我们就完成了按成员 j 降序排列的操作。和前面的 skip(10).limit(10) 的结果相比，j 的值是不同的。由于 sort 方法是对整个查询结果进行排序，因此对于查询结果来说，这些方法的执行顺序和实际的调用顺序无关，总是按照① sort ② skip ③ limit 的顺序来执行。

数据的更新和删除

只有文档的插入和查询并不能构成数据库的完整功能，我们还需要进行更新和删除。文档的插入我们使用了 save 方法，保存好的文档会被赋予一个 _id 成员，因此，当要保存的文档的 _id 已存在时，就会覆盖相应 _id 的文档。

也就是说，用 find 或 findOne 方法取出文档后，对取出的文档（JavaScript 对象）进行修改并再次调用 save（只有 _id 成员是不能修改的）的话，就会覆盖原来的文档。

在 MongoDB 中不存在事务的概念，因此总是以最后写入的数据为准。MySQL 在最开始不支持事务的时候还是非常有用的，由此可见，Web 应用中的数据库系统，即便不支持事务，貌似也不是很大的问题。MongoDB 中虽然不支持事务，但可以支持原子操作（atomic operation）和乐观并发控制（optimistic concurrency control）。要实现原子操作和乐观并发控制，可以使用 update 方法。

update 所支持的原子操作如表 3 所示，原子操作的名称都是以“$”开头的。例如，要将 j 为 0 的文档的 x 值增加 1，可以写成下面这样：

```
> db.bench.update({j:0},{$inc:{x:1}}) ⏎
```

表3　update的原子操作

名　　称	语　　法	功　　能
$inc	{$inc: {mem: n}}	对mem的值加n
$set	{$set: {mem: val}}	将mem的值设置为val
$unset	{$unset: {mem: 1}}	删除mem
$push	{$push: {mem: val}}	在数组mem中添加val
$pushAll	{$pushAll: {mem: ary}}	在数组mem中添加ary的元素
$addToSet	{$addToSet: {mem: val}}	当数组mem中不包含val时则添加val
$pop	{$pop: {mem: 1}}	删除数组mem中最后一个元素
$pop	{$pop: {mem: -1}}	删除数组mem中第一个元素
$pull	{$pull: {mem: val}}	从数组mem中删除所有的val
$pullAll	{$pullAll: {mem: ary}}	从数组mem中删除所有ary中的元素

乐观并发控制

然而，当需要进行并发操作时，仅凭原子操作还不够。在关系型数据库中，一般是通过事务的方式来处理的，但 MongoDB 中没有这样的机制。MongoDB 中进行并发操作的步骤如下。

(1) 通过查询获取文档。

(2) 保存原始值。

(3) 更新文档。

(4) 将原始值（包含 _id）作为第一参数，将更新后的文档作为第二参数，调用 update 方法。如果文档已经被其他并发操作修改时，则 update 失败。

(5) 如果 update 失败则返回第 (1) 步重新执行。

这样的方式，也就是利用了 update 方法可以进行原子更新这一点，通过同时指定事务开始时和更新后的值，来手动实现相当于关系型数据库中事务处理的功能。这种方法的前提，是基于“同一个文档基本上不会被同时修改”这一预测，也就是一种乐观的（近似的）事务机制。

需要显式创建数据的副本这一点有些麻烦，但忽略这一点的话，实际上还是很实用的。作为一个简单的例子，我们将刚才讲过的 $inc 那个例题，用乐观并发处理进行实现，如图 6 所示。

```
> for (;;) {
... var d = db.bench.findOne({j:0})
... var n = d.x
... d.x++
... db.bench.update({_id:d._id, x:n}, d)
... if (db.$cmd.findOne({getlasterror:1}).updatedExisting) break
... } ⏎
```

图 6　乐观并发处理

5.3 用 Ruby 来操作 MongoDB

关系型数据库为了保持其基本的 ACID 原则（原子性、一致性、隔离性、持久性），需要以付出种种开销作为代价。而相对地，MongoDB 这样的面向文档数据库由于可以突破这一局限，因此工作起来显得比较轻快。

MongoDB 具有下列这些主要特点：

- 以 JSON（JavaScript Object Notation）格式保存数据
- 不需要结构定义
- 支持分布式环境
- 乐观的事务机制
- 通过 JavaScript 进行操作
- 支持从多种语言进行访问

MongoDB 最重要的特点就是不需要结构定义。很少有应用程序在开发之前就能确定数据库中需要保存的数据项目。由于开发过程中的疏漏，或者是需求的变化等，经常导致数据库结构在开发中发生改变。在关系型数据库（RDB）中，遇到这种情况每次都需要重做数据库。Ruby on Rails 中可以通过 migration 方法对 RDB 结构迁移提供支持，但即便如此，这个过程依然相当麻烦。

而 MongoDB 本来就没有结构定义，即便数据库中保存的项目发生变化，只要程序做出应对就可以了。当然，已经存在的数据中不包含新增的项目，但要做出应对也很容易。

使用 Ruby 驱动

MongoDB 的另一个特点，就是可以由多种语言进行访问。为各种语言访问 MongoDB 所提供的库被称为驱动（driver）。MongoDB 分别为 JavaScript、C++、C#、Java、JVM 语言、Perl、PHP、Python 和 Ruby 提供了相应的驱动。

MongoDB 的服务器中集成了 JavaScript 解释器，因此回调等服务器端的处理只能用 JavaScript 来编写。不过，因为有了支持各种语言的驱动，客户端（除了发送给服务器的程序以外）

则可以用自己喜欢的语言来编写。

在 5.2 中我们使用 mongo 命令访问数据库，并使用 JavaScript 对数据库进行操作，不过如果可以用我们所习惯的 Ruby 来操作数据库就好了。RubyGems 中提供了相应的 Ruby 驱动，使用 gem 命令就可以轻松完成安装（以下命令均以 Debian 为例）：

```
% sudo gem install mongo ↵
```

此外，最好也一并安装用于加速访问的 C 语言库[①]，通过这个库可以提升与 MongoDB 服务器通信需要用到的“二进制 JSON”（BSON）的处理速度。

```
% sudo gem install mongo_ext ↵
```

要使用 MongoDB 的 Ruby 驱动，需要在程序中对 mongo 库进行 require。此外，在 Ruby 中还需要在 mongo 库之前对 rubygems 库进行 require。

```
require 'rubygems'
require 'mongo'
```

好了，我们来尝试访问一下 5.2 中创建的数据库服务器吧。首先我们需要创建一个表示服务器连接的 Mongo::Connection 对象。

```
m = Mongo::Connection.new
=> #<Mongo::Connection>
```

在后面的程序示例中，“=>”后面的部分表示表达式的求值结果。返回值的表示是以 irb 为基准的，但由于版面所限进行了大量的省略。此外，相当于 ID 的数值（包括数字位数在内）也会和实际情况有所不同。Mongo::Connection.new 可以带两个可选参数：第一个是主机名，第二个是端口号。这相当于 mongo 命令中的“--host”和“--port”参数。

通过这个连接，我们来尝试获取服务器所管理的数据库清单。

```
m.database_names
=> ["local", "admin", "test"]
```

要删除一个数据库，可以对数据库连接对象调用 drop_database 方法。

```
m.drop_database('test')
=> {"dropped"=>"test.$cmd",
    "ok"=>1.0}
```

① 由于 mongo_ext 是二进制 gems，因此需要另外安装开发环境（编译器以及编译用的库文件）。（原书注）

对数据库进行操作

对数据库连接调用 db 方法可以获得一个数据库对象，但在创建数据库对象的时间点上，实际上还并没有真正创建数据库。

```
db = m.db("nikkei_linux")
=> #<Mongo::DB>
m.database_names
=> ["local", "admin", "test"]
```

通过调用数据库对象的 collection 方法,可以获取相应的集合(相当于关系型数据库中的表)。如果要获取的集合不存在，则会创建一个新的集合。但是，和数据库一样，实际的集合也是要等到真正插入数据的时候才会被创建。

```
coll = db.collection("articles")
=> #<Mongo::Collection>
db.collection_names
=> []
```

数据的插入

使用 insert 方法或者 save 方法可以向集合中插入数据。

```
coll.insert({
  :title => "技术的剖析",
  :author => "matz"})
=> ObjectID('4bbf93')
```

当插入数据时，数据库和集合才会真正被创建出来。

```
m.database_names
=> ["local", "admin", "nikkei_linux"]
db.collection_names
=> ["articles", "system.indexes"]
```

这样，我们就创建了 nikkei_linux 数据库和 articles 集合。system.indexes 集合是 MongoDB 用于查询索引的集合。

数据的查询

当然，数据不光要能够插入，还要能够取出。当需要仅取出一个文档时，可以使用 find_one 方法。

```
coll.find_one()
=> {"_id"=>ObjectID('4bbf93'),
    "title"=>"技术的剖析",
    "author"=>"matz"}
```

这里我们没有指定查询条件，因为这个集合里面本来就只有一个文档，所以用这条语句便取出了这个唯一的文档。

我们再来尝试一下从更多的数据中进行查询吧。首先，我们用 insert 对数据库填充一定量的数据。

```
coll = db.collection("bench")
1000000.times {|i|
 coll.insert({:x => 4, :j => i})
}
=> 1000000
```

这样我们就在 bench 集合中插入了 100 万个文档。下面我们来查询一下看看。

```
coll.find_one({:j => 999999})
 => {"_id"=>ObjectID('4bbf93'),
     "x"=>4, "j"=>999999}
```

在我的电脑上查询到这个结果用了差不多 1 秒的时间。因为要将 100 万个文档全部查询一遍，所以这个速度不是很快，于是我们来创建一个索引。

```
coll.create_index("j")
=> "j_1"
```

这样我们就对 j 这个成员创建了一个索引。再查询一次试试看，瞬间就可以得到结果，索引的效果是非常明显的。

如果用 find 方法代替 find_one 方法的话，就可以得到一个指向所有符合条件的文档的“游标”（cursor）对象。

```
coll.find({:j => 999999})
=> #<Mongo::Cursor>
```

高级查询

刚才我们所进行的查询，都是像“集合中所有文档”或者“字段满足一定条件的文档”等简单的情况，但实际的查询并非都如此简单。在关系型数据库中，可以使用 SQL 来指定条件进行查询，MongoDB 当然也可以。例如，SQL 查询：

```
SELECT * FROM bench WHERE x = 4
```

这样的查询条件，用 Ruby 可以写成：

```
coll.find({:x => 4})
=> #<Mongo::Cursor>
```

也可以进行大小比较，如 SQL 查询：

```
SELECT j FROM bench WHERE x >= 4
```

用 Ruby 可以写成：

```
coll.find({:x => {"$gte" => 4}})
 => #<Mongo::Cursor>
```

除了等于的比较以外，其他的比较都是用以“$”开头的操作符来表达的。MongoDB 中可以使用的比较操作符如表 1 所示。

表1　比较操作符

名　　称	语　　法	含　　义
$gt	{"$gt" => val}	大于 val
$lt	{"$gt" => val}	小于 val
$gte	{"$gte" => val}	大于等于 val
$lte	{"$lte" => val}	小于等于 val
$ne	{"$ne" => val}	不等于 val
$in	{"$in" => val}	包含 val
$nin	{"$nin" => val}	不包含 val
$mod	{"$mod" => [n, m]}	除以 n 的余数为 m
$all	{"$all" => ary}	包含 ary 的所有元素
$size	{"$size" => n}	数组长度为 n
$exists	{"$exists" => true}	存在
$exists	{"$exists" => false}	不存在
$not	{"$not" => cond}	否定条件
正则表达式	^foo	等正则匹配
$where	{"$where" => str}	将 str 作为 JavaScript 进行求值，用 this 来引用文档

当然，通过多个条件的组合，也可以编写出像“大于 2，小于 8”这样的条件。

```
coll.find(:j=>{"$gt"=>2,"$lt"=>8})
=> #<Mongo::Cursor>
```

find 方法的选项

find 方法可以通过第二个参数来指定一些查询选项。

```
coll.find({:x=>4},{:limit=>10})
```

这表示在查询结果中只取出 10 个结果的意思。在 Ruby 中，末尾的参数为 Hash 时可以省略花括号，因此上述 find 调用也可以写成下面的形式：

```
coll.find({:x=>4},:limit=>10)
```

而且，在 Ruby 1.9 中，当 Hash 以符号（symbol）为键时，也可以写成省略形式：

```
coll.find({:x=>4},limit:10)
```

find 方法的选项及其含义如表 2 所示。

表2　find方法的选项

选　　项	值	含　　义
:skip	整数	整数跳过指定数量的结果。
:limit	整数	整数只取出指定数量的结果。
:sort	文字列	字符串按指定字段进行排序。
:sort	配列	数组按[字段,顺序]的格式指定排序条件。指定顺序时，升序为 :asc，降序为 :desc。
:fields	配列	数组指定结果文档中要包含的字段名。
:hint	文字列	字符串使用指定字段的索引进行查询。
:snapshot	真伪值	布尔值是否对文档进行快照。
:timeout	TRUE	TRUE 是否超时。当指定 :timeout 时，可附带代码块调用 find。

在 find 方法的选项中，skip、limit、sort 也可以作为 find 所返回的游标对象的方法来进行调用。因此，像：

```
coll.find({:x=>4},limit:10)
```

也可以写成：

```
coll.find(:x=>4).limit(10)
```

sort 在调用时可以不在数组中指定顺序，而是在参数中指定。因此，像：

```
coll.find({:x=>4},sort:[:j,:desc])
```

用游标方法的形式，也可以写成：

```
coll.find(:x=>4).sort(:j,:desc)
```

这两种写法，在内部处理上是完全相同的，因此选一种自己喜欢的写法就可以了。

原子操作

只有文档的插入和查询并不能构成数据库的完整功能，我们还需要进行更新和删除。文档的插入我们使用了 save 方法，保存好的文档会被赋予一个 _id 成员，因此，当要保存的文档的 _id 已存在时，就会覆盖相应 _id 的文档。也就是说，用 find 或 find_one 方法取出文档之后，将文档内容进行改写，然后再重新 save 的话（唯独不能改变 _id 成员的值），就可以替换原来的文档了。

MongoDB 中不支持事务机制，对于其他连接对同一文档进行更新的行为，是无法做出保护的。MySQL 在最开始不支持事务的时候还是非常有用的，由此可见，Web 应用中的数据库系统，即便不支持事务，貌似也不是很大的问题。

MongoDB 中虽然不支持事务，但可以通过 update 方法，在更新文档时排除来自其他连接的干扰。update 方法与文档的更新操作是互斥的，其操作结果只有“更新成功”和由于某些原因“出错失败”这两种状态。也就是说，当多个连接同时对同一个文档进行 update 操作时，更新操作也不会发生“混淆”，而是保证其中只有某一个操作能够成功。失败的操作可以进行重试，从结果来看，和按顺序执行更新操作是一样的。

像这样，更新操作不会半路中断，也不会留下不完整状态的操作，被称为“原子操作”。

update 方法最多可以接受三个参数。第一个是原始文档，第二个是新文档，最后一个是选项。其中选项是可以省略的。原始文档指的是更新之前的文档,但这里并不需要给出完整的文档，而是写成和 find 方法查询相同的格式即可。新文档指的是更新后的文档，这里也不需要给出完整的文档，只要给出包含更新后字段的 Hash 即可。当给出的字段已存在时就会更新其中的值，否则，就会添加一个新的字段。

update 方法的选项和 find 方法一样，是通过 Hash 来指定的。update 方法的选项如表 3 所示。我们用 update 方法，来进行对一个文档中的成员的值累加 1 这样的原子操作（图 1）。在图 1 中，如果在调用 update 的地方用 save 方法来代替，在取出文档到累加并保存的这段时间内，如果该文档被其他连接改写，累加操作就会失效。以图 2 为例，两个连接几乎同时进行累加操作，但由于取出和保存文档的顺序是混杂的，因此虽然进行了两次累加 1 的操作，但实际上 x 的值只增加了 1。

表3 update方法的选项

选 项	值	默认值	含 义
:upsert	布尔值	假	布尔值假当与“原始文档”相匹配的文档不存在时，则创建新文档。
:multi	布尔值	假	布尔值假当“原始文档”匹配到多个文档且该选项为真时，则更新所有文档。为假，则只更新其中一个文档。
:safe	布尔值	假	布尔值假为真时，会对是否真的完成了更新进行确认

```
loop
  doc = coll.find_one(:j=>0)  ◀——— 取出一个文档
  orig = doc.dup  ◀——— 将更新前的文档保存下来
  d["x"] += 1  ◀——— 更新字段x
  r = coll.update(orig, doc, :safe=>true)  ◀——— 调用update。不可以用coll.save(doc),为了确
                                                 认更新结果设置了:safe选项
  if r[0][0]["n"] == 1  ◀——— 更新成功则跳出循环
    break
  end  ◀——— 循环:返回开头。从取出文档的步骤开始重试
end
```

图 1 乐观并发控制

相对地，像图 1 这样使用 update 方法的话，在进行更新操作时，就会像图 3 一样发现文档被改写这一情况，并通过重试最终得到正确的结果。

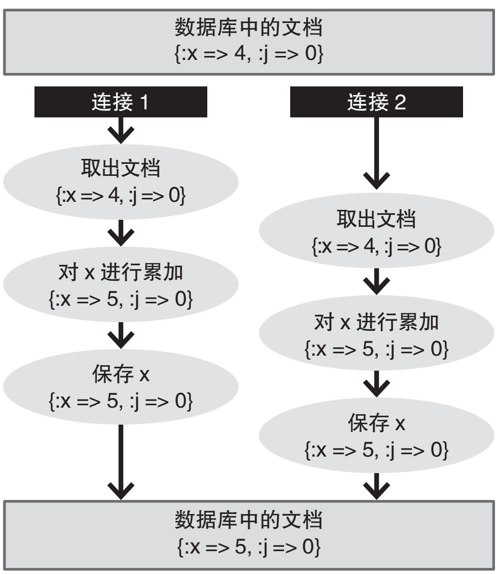


图 2 失败的并发控制

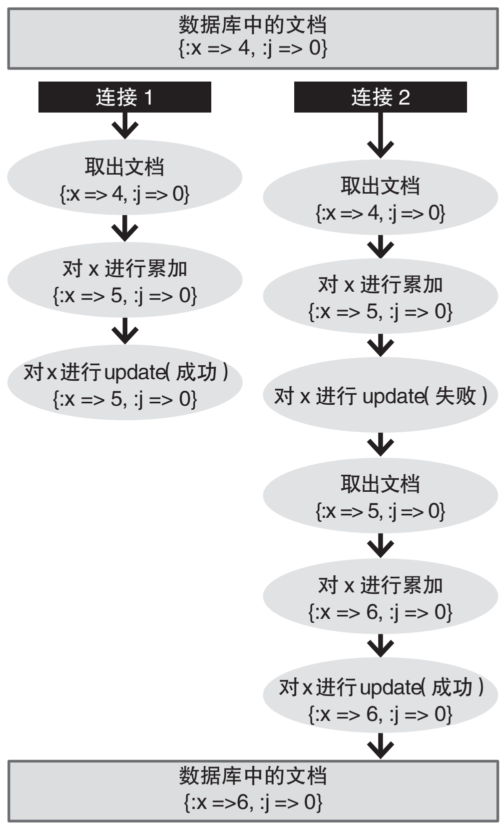


图 3 成功的并发控制

不过，像累加这样的典型操作，要是能编写得更简单一些就好了。其实，只要在 update 中对更新文档的指定上稍微优化一下，就可以用一个 update 语句实现某些典型的原子操作了。

例如，图 1 中的累加操作，也可以写成下面这样：

```
coll.update({:j=>0},
          {"$inc"=>{:j=>1}})
```

这行代码的意思是："找到字段 j 为 0 的文档，然后将 j 的值加 1"。

update 方法中可以使用的原子操作如图 4 所示。原子操作的名称都是以"$"开头的。表 3 中的"f"表示字段名，字段名可以通过字符串或者符号的形式进行指定。

表4　update的原子操作

名　称	功　能	语　法
$inc	对 f 的值加 n	{"$inc" => {f => n}}
$set	将 f 的值设为 val	{"$set" => {f => val}}
$unset	删除 f	{"$unset" => {f => 1}}
$push	在 f 所指的数组中插入 val	{"$push" => {f => val}}
$pushAll	在 f 所指的数组中插入 ary 的元素	{"$pushAll" => {f => ary}}
$addToSet	当 f 所指的数组中不存在 val 时则插入	{"$addToSet" => {f => val}}
$pop	删除 f 所指的数组的最后一个元素	{"$pop" => {f => 1}}
$pop	删除 f 所指的数组的第一个元素	{"$pop" => {f => -1}}
$pull	从 f 所指的数组中删除所有 val	{"$pull" => {f => val}}
$pullAll	从 f 所指的数组中删除 ary 的元素	{"$pullAll" => {f => ary}}

ActiveRecord

在 Ruby 的世界中，作为面向对象的数据库访问手段，最有名的莫过于 ActiveRecord 了。在 Ruby on Rails 中，对关系型数据库的操作并不是直接进行的，而是通过 ActiveRecord 库将表中的各条记录作为对象来进行操作的。像这种将对象与记录进行对应的库，被称为 OR Mapper。其中 O 代表对象（Object），R 代表关系（Relation），因此它就是将关系与对象进行映射的意思。

有了 ActiveRecord 的帮助，在 Rails 程序设计中，可以不必关心底层的关系型数据库，完全以面向对象的方式来进行编程。ActiveRecord 是一种十分易用的 OR Mapper,但也并非完美无缺。其中一个令人不满意的地方，就是数据库结构信息和模型①定义是分离的。由于 Rails 的组件遵

① 模型指的是 MVC（Model-View-Controller）架构中的 Model。

循 DRY(Don’t Repeat Yourself)原则，因此 ActiveRecord 中不会对数据库结构定义进行重复描述。数据库结构信息只存在于它原本存在的地方，也就是数据库中。

信息的重复往往是造成 bug 的元凶，从这一点上来看，DRY 原则是非常优秀的。但另一方面，对数据的操作以及对象之间的关系，则是在模型中进行定义的。从这个意义上来说，对象结构的信息和对其进行操作的信息是分开的。如果要查看字段信息，就必须要查看运行中的数据库。因此，想要将相关信息都整合在一起，是再自然不过的需求了。

另一个不满意的地方，则是 ActiveRecord 所提供的“一条记录 = 一个对象”这样的抽象化模型，并非总是最优的。在简单的水平上，“一条记录 = 一个对象” 这样的抽象化模型实现起来很容易，只要将 SQL 调用所得到的记录封装成对象就可以了。但是，当对象的调用变得越来越频繁和复杂时，就会产生性能上的问题。结果，关系型数据库中的记录，并没有成为真正意义上的对象，在特殊情况下，会暴露出抽象化中的纰漏。这样的问题，被称为抽象泄漏（ leaky abstraction ）。

如果为了改善性能而使用缓存等手段的话，模型的逻辑会变得越来越复杂。另一方面，为了得到更优化的 SQL，会对 find 方法等指定非常详细的选项，结果则是无法用 Ruby 来编写，最后变成直接写 SQL 了。这也许已经是 ActiveRecord 的极限了吧。

OD Mapper

这个时候，就轮到 MongoDB 发挥威力了。由于 MongoDB 是不需要数据库结构的，因此结构定义和模型定义分离的问题也就不存在了。此外，MongoDB 中也没有 SQL，原本也无法进行复杂的查询，因此也不必担心会写出“SQL 式的 Ruby”。当然，也许大家会觉得这样是治标不治本，不过这个问题本来就应该分开来看。也就是说，如果使用 MongoDB 的话（如果应用程序能够使用 MongoDB 来实现的话），对 ActiveRecord 的那些不满也就可以得到缓解了。

要在 Rails 中使用 MongoDB，有一些是可以与 ActiveRecord 互换的库。

- MongoMapper
- Mongoid
- activerecord-alt-mongo-adapter

由于 MongoDB 不是关系型数据库，因此这些库也不能称之为 OR Mapper，而是应该称之为 Object Document Mapper（OD Mapper）了吧。

1. MongoMapper

MongoMapper 是 MongoDB 用 ActiveRecord 驱动中资历最老的一个，作者是 John Nunemaker。话说，MongoDB 本身是 2009 年才诞生的，因此所谓资历最老其实也老不到哪里去呢。它与 ActiveRecord 之间的兼容性很高，在 Rails 应用程序中，即便将 MongoMapper 当成 ActiveRecord 的替代品来使用，各种 Rails 插件也都能正常工作。

```
require 'rubygems'
require 'mongo_mapper'

class Employee
  include MongoMapper::Document
  key :first_name
  key :last_name
  many :addresses
end

class Address
  include MongoMapper::EmbeddedDocument
  key :street
  key :city
  key :state
  key :post_code
end
```

图 4　用 MongoMapper 进行模型定义

MongoMapper 中的模型定义如图 4 所示。MongoDB 中对模型进行定义时，本来就不需要数据库结构定义，实际上，在 MongoMapper 中结构定义也不是必需的，但也可以通过 key 方法对字段及其类型进行显式的声明。通过这样的显式声明，可以利用类型检查，使得一些问题更容易被发现，同时也使得数据库结构能够文档化。刚才我们讲过对 ActiveRecord 的不满意的地方，而在这里，我们可以将数据库的结构和操作在同一个地方进行定义，感觉非常好。

在 MongoDB 中，可以在一个文档中嵌入另一个文档（JSON），这在一般的关系型数据库中是很难做到的[①]。当然，原本就是基于关系型数据库的 ActiveRecord 自然也不支持嵌入文档，而在 MongoMapper 中，对于嵌入的文档只要将 include 的类由 MongoMapper::Document 改成 MongoMapper::EmbeddedDocument，就表示该对象不是保存在独立的集合中，而是一个嵌入文档。

2. Mongoid

Mongoid 是一个比 MongoMapper 更新一些的库，作者是 Durran Jordan。Mongoid 的功能很丰富，通过和 ActiveRecord 类似的 API 可以充分发挥 MongoDB 的全部功能。用 Mongoid 定义一个和图 4 相同的模型，代码如图 5 所示。在一些细节上有所差别，但大意是一样的。不过，Mongoid 中是没有对于嵌入文档的定义的，但 Mongoid 中其实有一条规则：凡是指定了“inverse_of”的对象关系都会自动被视为嵌入。这一点还是相当智能的。

① 也有一些关系型数据库提供了可以直接引用另外一张表指定部分的功能。（原书注）

此外，对于 MongoDB 所提供的如支持文档的类继承、验证、版本控制、回调，以及基于 JavaScript 的 MapReduce 等功能，Mongoid 都通过和 ActiveRecord（以及 ActiveModel）相类似的 API 进行了实现，这也是 Mongoid 的设计思想之一。

```
require 'rubygems'
require 'mongoid'

class Employee
  include Mongoid::Document
  field :first_name
  field :last_name
  has_many :addresses
end

class Address
  include Mongoid::Document
  field :street
  field :city
  field :state
  field :post_code
  belongs_to :employee, :inverse_of =>
:addresses
end
```

图 5　用 Mongoid 进行模型定义

3. activerecord-alt-mongo-adapter

activerecord-alt-mongo-adapter 是最新的一个库，作者是 SUGAWARA Genki。MongoMapper 和 Mongoid 都是替代 ActiveRecord 来使用的库，相比之下，activerecord-alt-mongo-adapter 则是一个用于 ActiveRecord 的 DB 适配器(图 6)。换句话说，它并不是一个独立的替代库，而是一个通过 ActiveRecord 的数据库切换功能来使用的 MongoDB 访问适配器。因此，ActiveRecord 本身的功能都可以直接使用。仔细想想的话，ActiveRecord 虽然通过提供相应的适配器的方式实现了对各种数据库的支持，而且只要修改配置文件就可以对数据库系统进行切换，但其工作方式总归还是基于 SQL 的。

```
class Employee < ActiveRecord::Base
  include ActiveMongo::Collection
  has_many :addresses
end

class Address < ActiveRecord::Base
  include ActiveMongo::Collection
end
```

图 6　用 activerecord-alt-mongo-adapter 进行模型定义

然而，MongoDB 属于 NoSQL，当然是无法用 SQL 来进行解释的。为了搞清楚这个适配器是如何实现对 MongoDB 的支持的，我看了一下它的源代码。它为了能够解释从 ActiveRecord 传来的 SQL，居然用 Ruby 编写了一个 SQL 语法解析器，而对于 MongoDB 的访问是通过这个 SQL 语法解析器来完成的。唔，好厉害啊。

不过，以 ActiveRecord 适配器的形式工作也并非尽善尽美，在我查到的范围内，像对嵌入文档的支持、模型内部字段声明、在模型定义中创建索引等功能都是不支持的。这些功能在关系型数据库中本来就不存在，而 ActiveRecord 也原本就没有考虑到在非关系型数据库上进行应用，因此在这一点上也不能指望 ActiveRecord。即便如此，通过利用 ActiveRecord，它只用了相当于 MongoMapper 和 Mongoid 十分之一的代码量，就实现了对 MongoDB 的访问，可以说是很了不起的。

说到底，activerecord-alt-mongo-adapter 只是一个 ActiveRecord 适配器。因此，作为 ActiveRecord 的特点之一，它可以通过 database.yml 在开发环境 DB 和正式环境 DB 之间进行自动切换，这可以说是一个比较大的优点。

但与此同时，仅通过这个适配器很难用到 MongoDB 的全部功能，而且由于需要经过一个额外的 SQL 解析层，性能方面也很让人担心。

5.4 SQL 数据库的反击

有一种说法称：云计算不再是 SQL 的时代，而是 NoSQL 的时代。因此，不依赖 SQL 且结构简单的 NoSQL 数据库受到了广泛的关注。那么，SQL 数据库真的已经不再那么重要了吗？SQL 数据库真的不支持云计算吗？

“云”的定义

“云”这个词有很多种用法，不过它的定义却非常模糊，导致我们的讨论容易过度发散。为了让论点更加明确，我们在这里将“云”定义为“大规模分布式环境”这个概念。

当然，“云”并非总是代表“大规模分布式环境”，但在数据库系统的语境中，用“云”这个词一般是代表现有数据库系统很难应对的情况。也就是说，数据量和访问量这两者的其中一个，甚至是全部两个，其规模已经超过单独一台数据库服务器所能够应对的程度，必须要依靠由多台服务器协同工作所构成的分布式环境来进行应对。

在这样的环境中，不使用 SQL 的 NoSQL 数据库很受欢迎。像 SQL 这样复杂的查询访问是受限的，取而代之的则是多个 NoSQL 数据库自动分布到多台服务器上的架构。这种方式对大规模分布式环境拥有更强的亲和力。

SQL 数据库的极限

那么，SQL 数据库真的不适合大规模分布式环境吗？在云计算环境中，它真的就不如 NoSQL 数据库吗？事实上并不一定。在云计算环境下最大限度利用现有 SQL 数据库的技术，目前已经在实用化方面取得了一定的进展。

其中一个基本的思路是对数据库进行分割。在专业领域，这种数据库分割被称为 Sharding 或者 Partitioning。这种手法的目的，是通过将数据库中大量的记录分别存放到多台服务器中，从而避免数据库服务器的瓶颈。以 mixi 这样的社交网站（SNS）为例，可以理解为将用户编号为偶数的用户和编号为奇数的用户分别存放到不同的数据库中。这样就避免了对单独一台数据

库服务器的集中访问，从而提高了处理速度。第一步的分割可以在应用程序级别上完成，在上述例子中，对编号为偶数和奇数的记录分别访问不同的数据库，而这样的逻辑可以编写在应用程序中。

不过，仔细想想就会发现，其实数据库的分割和应用程序逻辑的本质毫无关系，是需要在数据库层面上解决的问题。将这样的逻辑混入应用程序中的话，说实话是很“拙劣”的。数据库的问题，就应该在数据库中解决，不是吗？像这样能够实现自动分割的方法有很多种，这里我们介绍一下为 MySQL 提供分割功能的 Spider。

存储引擎 Spider

Spider 是由 ST Global 公司的斯波健德（Kentoku Shiba）先生开发的一种存储引擎。在 MySQL 中，用于查询处理的数据库引擎和实际负责存储数据的存储引擎是相互独立的，对于每张数据表都可以采用不同的存储引擎。可能大家都听说过 InnoDB、MyISAM 之类的名字，这些都是 MySQL 的存储引擎。

Spider 和它们一样，也是在 MySQL 上工作的存储引擎的一种。不过，Spider 自身并不执行实际的数据存储操作，而是将这些操作交给其他的 MySQL 服务器来完成。也就是说，在使用 Spider 的时候，表面上看起来是一个数据库，实际上却可以将数据自动分割保存在多台数据库中（sharding），而且只要对一台数据库保存数据，也会同时在其他服务器的数据库中保存（replication）。比起在应用程序端实现分割来说，用 Spider 来实现有下列这些优点：

- **逻辑和数据库相分离**：使用 Spider，就意味着从应用程序端看起来，对数据库的访问和通常的 MySQL 访问是完全一样的。因此，在应用程序端不需要进行任何特殊的应对。
- **可维护性高**：和分割相关的信息都只维护在表定义中，而且，数据库分割策略也可以在表定义中进行设置。关于数据库的设置都集中在一个地方，这一点从可维护性的角度来说，是非常重要的。

刚才我们介绍了利用 MySQL 的存储引擎 Spider 进行自动分割的手法。其实实现自动分割的软件不仅只有这一种，单在 MySQL 数据库中，除了 Spider 之外，还有像 MySQL Cluster、SpockProxy 等其他方案。

SQL 数据库之父的反驳

尽管通过 Sharding 技术将数据库进行分割，就能够在分布式环境中运用 SQL 数据库，但却

无法做到像一部分 NoSQL 数据库那样，能够根据需要自动增加节点来实现性能的扩充。此外，如果用 SQL 数据库来实现 NoSQL 中这种简单的查询处理，大多数情况下在性能上（如每秒查询数）都不及 NoSQL。

虽说 SQL 和 NoSQL 各自所擅长的领域不同，但很多人曾经认为在大规模分布式环境中使用 NoSQL 是板上钉钉的事。在这个时候，迈克尔・斯通布雷克（Michael Stonebraker，1943—　）站了出来。斯通布雷克是最早的 RDB 系统 Ingres 的开发者，在 Ingres 商用化之后，他开发了 Ingres 的后续版本 Postgres，后者演变为现在的 PostgreSQL。斯通布雷克应该被称为 PostgreSQL 之父，但他的贡献并非仅仅如此。由于 Sybase 以及 Microsoft SQL Server 中都继承了他所开发的 Ingres 的代码，因此毫无疑问，他是一个对于 SQL 数据库整体都产生了巨大影响的人物。现在，斯通布雷克担任 MIT（麻省理工学院）客座教授，同时还在几家数据库相关的企业中担任董事。

斯通布雷克在计算机协会 ACM①的学术期刊《Communications of the ACM》（ACM 通信）2010 年 4 月号中刊登了一篇题为“SQL Databases vs NoSQL Databases”的专栏②。在该专栏中，斯通布雷克以“所有的技术都有其擅长的领域，没有一种数据库是万能的”为前提，提出了以下观点：

- NoSQL 的优势在于性能和灵活性。
- NoSQL 的性能优于 SQL 这一说法，并非在所有情况下都成立。
- 通常认为 NoSQL 是通过牺牲 SQL 和 ACID 特性③来实现其性能的，然而性能问题与 SQL 和 ACID 是无关的。

说实话，看了这些内容，我的第一反应就是：“唉？真的吗？”。作为像我这样写了很多文章，给别人灌输了“云计算时代非 NoSQL 莫属”观点的人来说，实在是百思不得其解。

那么我们就来看个究竟吧。根据这篇文章，决定 SQL 数据库性能的，是客户端与服务器之间的通信开销，以及服务器上的事务处理开销。而通信开销可以通过将大部分处理放在服务器上的存储过程（Stored Procedure）在一定程度上得以解决。

① 计算机协会（Association of Computer Machinary，ACM）是一个世界性计算机从业者的专业组织，被誉为计算机领域的诺贝尔奖的图灵奖就是由 ACM 主办的。

② 斯通布雷克的专栏可参见：http://cacm.acm.org/blogs/blog-cacm/50678-the-nosql-discussion-has-nothing-to-do-with-sql/fulltext。（原书注）

③ 数据库所应具备的 4 种特性的首字母缩写，即一系列处理不会残留中间状态的 Atomicity（原子性）、不会产生数据不完整的 Consistency（一致性）、对处理中间状态进行隐藏的 Isolation（隔离性）以及对完成的处理进行保存的 Durability（持久性）。（原书注）

而对于服务器上的处理，大致进行分类的话，主要有 4 个瓶颈，而对于这些瓶颈的应对就是决定性能的关键。这 4 个瓶颈具体如下。

日志（Logging）：为了防止磁盘崩溃等故障的发生，大多数关系型数据库都会执行两次写入。即向数据库执行一次写入，再向日志执行一次写入。而且，为了防止日志信息丢失（为了实现 ACID 中的 D），必须保证这些数据确实写入了磁盘中。这样，即便由于一些问题导致数据库崩溃，也可以根据日志的内容恢复到故障前的状态。然而，考虑到向磁盘写入的速度是非常慢的，因此向日志执行确定的写入操作是非常“昂贵”的。

事务锁（Locking）：在对记录进行操作之前，为了防止其他线程对记录进行修改，需要对事务加锁。这也形成了一项巨大的开销。

内存锁（Latching）：Latch 是门闩的意思，这里是指对锁和 B 树等共享数据结构进行访问时所需要的一种排他处理方式，斯通布雷克管这种方式叫做 Latching。这也是造成开销的原因之一。

缓存管理（Buffer Management）：一般来说，数据库的数据是写入到固定长度的磁盘页面中的。对于哪个数据写入哪个页面，或者是哪个页面的数据缓存在内存中，都需要由数据库进行管理。这也是一项开销很大的处理。

斯通布雷克认为，要实现高速的数据库系统，必须要消除上述所有 4 个瓶颈，而且上述瓶颈并非 SQL 数据库所固有的。听他这么一说，好像还真是这么回事。

NoSQL 之所以被认为速度很快，是因为它在设计之初就考虑了分布式环境，通过多个节点将处理分摊了。然而，SQL 数据库也是可以将处理分摊到多个节点上的。此外，即便是 NoSQL 数据库，只要涉及到磁盘写入操作，以及多线程架构下的缓存管理，也难以回避上述瓶颈中的一个或几个。

通过上面的分析，斯通布雷克的结论是，无论是 SQL 还是 ACID 特性，都不是影响云计算环境下数据库性能的本质性障碍。唔，原来如此。

随后，斯通布雷克还写了一篇博客[①]，对于与 CAP 原理[②]相对应的 NoSQL 数据库策略 BASE（Basically Available, Soft-state, Eventually consistent）阐述了反对意见。这还真是让人大开眼界。

① 网址：http://cacm.acm.org/blogs/blog-cacm/83396-errors-in-database-systems-eventual-consistency-and-the-cap-theorem/fulltext（原书注）

② 在“一致性”、“可用性”和“分裂容忍性”三者中，只能同时满足其中两者。据说这一原理已经通过数学方法得到了证明。而 BASE 是以满足“可用性”和“分裂容忍性”为目标的。（原书注）

SQL 数据库 VoltDB

之前讲的这些，还只是停留在“SQL 数据库在理论上还存在着可能性”这个阶段，但作为技术大牛的斯通布雷克可不会仅仅满足于这样的结论，他已经在数据库业界的最前线活跃了 40 多年，绝对不是一个简单的人物。斯通布雷克在美国一家叫做 VoltDB 的创业型公司任 CTO，该公司将他的上述观点进行了体现，开发出了 VoltDB 数据库系统，并以开源形式发布。这是何等惊人的行动力。

VoltDB 有两个版本，一个是以 GPL 协议发布的开源社区版本（Community Edition），另一个是以订阅形式提供的收费版本。开源版本可以直接从 VoltDB 公司的官方网站获取。

VoltDB 并不是一个像 PostgreSQL 或 MySQL 那样的通用数据库，而是一个面向特定领域（OLTP）进行了大幅度调优的数据库系统。在 VoltDB 的主页上是这样介绍它的：“VoltDB 是面向大规模事务处理应用程序的 SQL 数据库系统。”其特征包括以下几点：

- 比传统 RDBMS 高出几十倍的性能
- 线性可扩展性
- 以 SQL 作为 DBMS 接口
- ACID 特性
- 可 365 天 24 小时全天候工作的高可用性

看起来很有吸引力对吧？

不过，到底是用了怎样的手段才实现了这样的特性呢？尤其是斯通布雷克在博客中指出的那四个瓶颈，到底是用什么办法来解决的呢？

首先，VoltDB 最大的特征在于，它是一个内存数据库系统。也就是说，数据基本上是储存在内存中的。由于数据存储在内存中，缓存管理的问题就得以解决，而且由于不存在磁盘崩溃的情况，也就不需要日志了。这样一来，四个瓶颈中一下子就解决了两个。

等等，先别高兴太早。电源一关，内存中的数据就消失了，这样的数据库岂不是无法提供 ACID 中的持久性这一特性吗？其实，在 VoltDB 中，持久性是通过复制（replication）的方式来维持的。VoltDB 数据库是在由多台服务器组成的集群上工作的，在集群中的多台服务器上都保存有重复的数据副本，因此即便失去一台服务器，也不必担心数据会丢失。此外，VoltDB 也提供了定期将数据写入文件的快照功能。

那么，剩下的两个瓶颈，即事务锁和内存锁又是如何解决的呢？在 VoltDB 中，数据库是分

割成多个分区（partition）来管理的，对于每个分区都分配了一个独立的管理线程。也就是说，对分区的操作是单线程的，因此也就从根本上不需要用于实现排他处理的事务锁和内存锁机制了。

VoltDB 通过这样的构造回避了瓶颈，其每秒事务数（TPS）可以跑出比传统 RDBMS 高 50 倍的成绩（根据 VoltDB 公司的测试数据）。怎么说呢，这是拿一般的磁盘写入型数据库和内存型数据库来比较，有这种程度的差距也许并不意外。此外，由于上述比较是在一台服务器上进行的，而 VoltDB 可以通过增加节点来使性能呈线性提升，也就是说，如果将服务器数量翻倍，则性能也几乎可以翻倍，因此还是非常值得期待的。

VoltDB 的架构

VoltDB 采用以内存型集群运用为前提的架构，光这一点就和传统的 RDBMS 架构大相径庭，但它们之间的差异还远远不仅限于此。没有人知道，依靠打破 RDBMS 常识的想象力，向超高速的实现发起挑战的斯通布雷克，在这条路上到底能走多远。

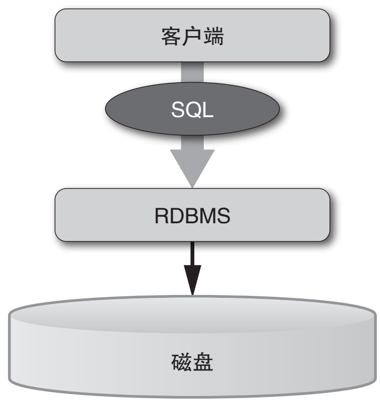


图 1　传统的 RDBMS 架构

大家应该还记得，在上述“4 个瓶颈”之前，曾经讲过“通信开销过大的问题，通过存储过程可以在一定程度上得到解决”。在传统的数据库中，有专门负责数据存储的数据库服务器，而应用程序是通过 SQL 来进行查询的（图 1）。而对于需要重复进行的操作，可以通过某种手段，调用服务器上事先写好的存储过程来实现，从而在一定程度上减少通信量。存储过程的实现方式在各种 RDBMS 上都有所不同，有的是用 C 等语言来编写并载入到服务器上，也有的是将 SQL 进行扩展来编写存储过程。

不过，“走极端”的 VoltDB 可不是光有存储过程就满足了的。“既然存储过程可以改善性能，那么把所有的事务都用存储过程来实现不就好了吗？”于是，在 VoltDB 中，对服务器的调用只能运行事先编写好的过程。也就是说，VoltDB 虽然是一个 SQL 数据库，但却无法从客户端来执行 SQL 查询（实际上貌似是可以的，只是不推荐而已）。这是何等的大刀阔斧。从结果来看，对 VoltDB 的访问不能使用现在主流的 ODBC 和 JDBC 方式，因为这些方式都是通过 SQL 来调用 RDBMS 功能的。要对 VoltDB 进行访问，需要用 Java 来编写程序。

也就是说，在 MySQL 等现有 RDBMS 的结构中包括通用的数据库服务器，客户端用 SQL 对该服务器进行访问，而在 VoltDB 中，访问数据库的过程本身是作为存储过程保存在数据库服务器中的，而客户端采用一种类似远程调用的方式对该存储过程进行调用（图 2）。换句话说，数据库服务器与客户端的界限，在位置上有所不同。

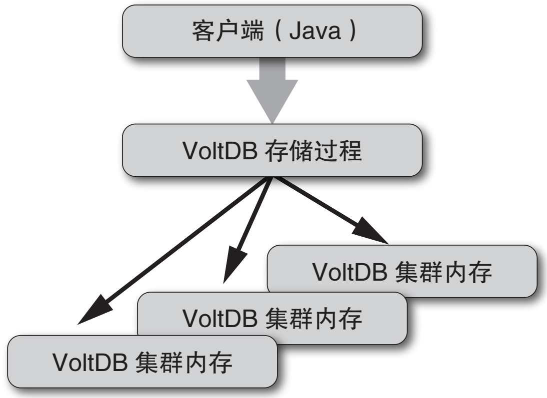


图 2　VoltDB 架构

VoltDB 中的编程

接下来，我们来看看在这一极端设计原则下，VoltDB 的编程到底是如何进行的。

VoltDB 的应用程序，基本上是采用图 3 这样的结构。要构筑一个应用程序，需要准备下列部件。

(1) 数据库结构定义：用于定义数据库的 SQL 文件，内容基本上是 SQL 的 CREATE TABLE 语句。此外，CREATE INDEX 和 CREATE VIEW 也是可以使用的。

(2) 存储过程：用 Java 编写的存储过程。关于存储过程的内容我们稍后会详细讲解，基本上是负责构造 SQL 语句并执行它。当然，由于存储过程采用 Java 来编写，自然可以在服务器端执行更加复杂的逻辑。

(3) 工程定义：一个名为 project.xml 的 XML 文件。在这个文件中定义了数据库定义文件名、存储过程名、数据库分割基准字段等信息。

(4) 客户端代码：客户端的源代码，用 Java 编写。对 VoltDB 的访问可以使用 org.voltdb 包所提供的功能。

将上述 (1) 至 (3) 的信息提交给一个叫做 VoltCompiler 的程序，就可以生成叫做“目录”（catalog）的服务器端程序（jar 文件）。此外，VoltCompiler 中还需要指定下列内容：

(a) 构成集群的节点数量
(b) 每个节点的分区数量
(c) 集群“首领”的主机名

上述信息需要在编译时指定，这意味着在 NoSQL 中十分常见的在运行时根据需要添加节点的动态可扩展性，这在 VoltDB 中是无法实现的。

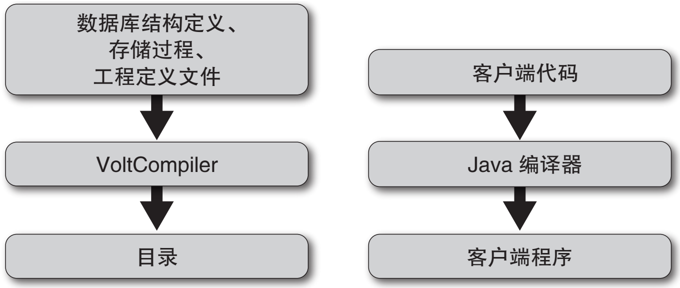


图 3　VoltDB 应用程序结构

在文档中，关于改变集群节点数量的步骤是这样写的：

① 将数据库输出至文件
② 修改节点数并重新生成目录
③ 重新启动数据库
④ 从文件向数据库载入数据

如果要对数据库结构定义进行修改，也需要按照上面的步骤来进行操作。

如果你习惯了 MongoDB 等 NoSQL 数据库的灵活性，就会觉得仅仅是为了修改结构定义和集群节点数就要这样操作实在是太麻烦了。不过与此同时，文档中也写道："VoltDB 在 2 到 12 个节点的环境下能够发挥最大效率，用 10 个节点就可以实现 100 万个 QPS（每秒查询数量），大多数情况下这样的性能已经可以满足需要"。也就是说，用少量的节点数量就能实现压倒性的性能，因此灵活性也就显得不那么重要了。从某种意义上说，这样的思路和 NoSQL 正好是相互对立的两个极端。

Hello VoltDB!

下面我们来看看实际的 VoltDB 程序。图 4 到图 8 就是 VoltDB 的 Hello World 程序。具体来说，就是用 Insert 存储过程将语言名称和该语言对应的 HelloWorld 写入数据库，然后用 Select 存储过程读出语言名称对应的 HelloWorld。

图 4 的数据库结构定义是一个简单的 SQL 中的 CREATE TABLE（创建表）语句，没有什么难点吧。图 5 和图 6 是用 Java 编写的存储过程定义。

```
CREATE TABLE HELLOWORLD (
   HELLO CHAR(15),
   WORLD CHAR(15),
   DIALECT CHAR(15) NOT NULL,
   PRIMARY KEY (DIALECT)
);
```

图 4　数据库结构定义（helloworld.ddl）

VoltDB 的存储过程定义包括下列要素：

- ❑ 对 org.voltdb 包的 import
- ❑ 用 @Procinfo 指定分区

- 定义一个继承 VoltProcedure 的类
- 定义存储过程本体的 run 方法

其中，有必要对 @ProcInfo 部分做一些说明。VoltDB 是将数据库分割成分区，并分布式地配置在集群中的节点上的。当存储过程所进行的操作仅限于对单一分区进行访问时，VoltDB 可以发挥最佳性能。因此，如果要在编译时指定某个存储过程是否仅访问单一分区，以及该分区的分割基准是哪张表的哪个字段，就需要使用 @ProcInfo 记法。

project.xml 中记载了关于数据库结构、存储过程、分区等信息。

```
import org.voltdb.*;

@ProcInfo(
    partitionInfo = "HELLOWORLD.DIALECT: 0",
    singlePartition = true
)

public class Insert extends VoltProcedure {

  public final SQLStmt sql = new SQLStmt(
      "INSERT INTO HELLOWORLD VALUES (?, ?, ?);"
  );

  public VoltTable[] run(String hello,
                         String world,
                         String language)
      throws VoltAbortException {
          voltQueueSQL(sql, hello, world,
language);
          voltExecuteSQL();
          return null;
      }
}
```

图 5　Insert 存储过程（Insert.java）

```
import org.voltdb.*;

@ProcInfo(
    partitionInfo = "HELLOWORLD.DIALECT: 0",
    singlePartition = true
)

public class Select extends VoltProcedure {

  public final SQLStmt sql = new SQLStmt(
      "SELECT HELLO, WORLD FROM HELLOWORLD " +
      " WHERE DIALECT = ?;"
  );

  public VoltTable[] run(String language)
      throws VoltAbortException {
          voltQueueSQL(sql, language);
          return voltExecuteSQL();
      }
}
```

图 6　Select 存储过程（Select.java）

```
<?xml version="1.0"?>
<project>
    <database name='database'>
       <schemas>
           <schema path='helloworld.sql' />
       </schemas>
       <procedures>
           <procedure class='Insert' />
           <procedure class='Select' />
       </procedures>
       <partitions>
           <partition table='HELLOWORLD'
column='DIALECT' />
       </partitions>
    </database>
</project>
```

图 7　工程定义（project.xml）

客户端代码则比较简单。基本上，对数据库的访问，只是使用 callProcedure() 方法，对 project.xml 中定义的存储过程进行调用而已。关于获取结果等操作，看一下图 8 中的示例代码，应该就能有个大致的理解了。

```
import org.voltdb.*;
import org.voltdb.client.*;

public class Client {

    public static void main(String[] args) throws Exception {

        /*
         * Instantiate a client and connect to the database.
         */
        org.voltdb.client.Client myApp;
        myApp = ClientFactory.createClient();
        myApp.createConnection("localhost", "program", "password");

        /*
         * Load the database.
         */
        myApp.callProcedure("Insert", "Hello", "World", "English");
        myApp.callProcedure("Insert", "Bonjour", "Monde", "French");
        myApp.callProcedure("Insert", "Hola", "Mundo", "Spanish");
        myApp.callProcedure("Insert", "Ciao", "Mondo", "Italian");
        myApp.callProcedure("Insert", "こんにちわ", "世界", "Japanese");

        /*
         * Retrieve the message.
         */
        final ClientResponse response = myApp.callProcedure("Select",
                                                            "Spanish");
        if (response.getStatus() != ClientResponse.SUCCESS){
            System.err.println(response.getStatusString());
            System.exit(-1);
        }

        final VoltTable results[] = response.getResults();
        if (results.length != 1) {
            System.out.printf("I can't say Hello in that language.");
            System.exit(-1);
        }

        VoltTable resultTable = results[0];
        VoltTableRow row = resultTable.fetchRow(0);
        System.out.printf("%s, %s!¥n", row.getString("hello"),
                                       row.getString("world"));
    }
}
```

图 8　客户端代码（Client.java）

性能测试

在 VoltDB 的官方网站（https://voltdb.com/blog/key-value-benchmarking）中，刊登了与 NoSQL 的代表 Cassandra 进行对比的性能测试结果。不过，需要注意的是，VoltDB 和 Cassandra 在擅长

处理的对象方面是有所不同的。

例如，VoltDB 是内存型数据库，Cassandra 则不是。Cassandra 无需定义数据库结构，比较灵活，VoltDB 则不是。在对比中需要注意上述这几点。

VoltDB 和 Cassandra 在以下三种性能测试中进行了对比。

单纯键 – 值：VoltDB 具备用于实现键 – 值存储所需的充足性能，因此对用 VoltDB 实现的键 – 值存储与 Cassandra 进行了对比。测试内容为用 50B 的键和 12KB 的值进行 50 万对的访问和更新。在单节点、3 节点（无复制）和 3 节点（有复制）这三种条件下对比 5 分钟的处理总数。

多个整数列：用 50 个 32 位整数作为值来代替单纯键 – 值进行对比。对比条件依然是单节点、3 节点（无复制）和 3 节点（有复制）三种。

多个整数列（批处理）：依然是用 50 个 32 位整数作为值，但不同的是对每个键进行 10 次访问和更新。对比条件依然是单节点、3 节点（无复制）和 3 节点（有复制）三种。

上述 3 次测试的结果如表 1 至表 3 所示。虽然集群的构成只有 3 个节点，规模比较小，但在对比的范围内，最好的情况下可以跑出相当于 Cassandra 性能 16 倍以上的成绩。不过，这个测试的目的，是为了证明使用 SQL 的数据库在云计算环境下也并不慢，而并不代表对大家接下来开发的应用程序数据库来说，VoltDB 就是最佳选择，这一点请大家注意。

表1　键–值性能测试（5分钟事务数量）

集群配置	VoltDB	Cassandra	性　能　比
1节点	17000	7940	2.2倍
3节点（无复制）	19800	17400	1.1倍
3节点（有复制）	12600	4450	2.8倍

表2　多个整数列性能测试（5分钟事务数量）

集群配置	VoltDB	Cassandra	性　能　比
1节点	111000	24200	4.6倍
3节点（无复制）	293000	38900	7.5倍
3节点（有复制）	176000	24700	7.1倍

表3　多个整数列（批处理）性能测试（5分钟事务数量）

集群配置	VoltDB	Cassandra	性　能　比
1节点	102000	13300	7.7倍
3节点（无复制）	286000	17200	16倍
3节点（有复制）	172000	13000	13倍

小结

VoltDB 是一种内存型数据库，且客户端无法直接调用 SQL，这与传统的 RDBMS 在设计思想上有很大差异。然而相对地，它却展现出了优秀的性能，并可以通过增加节点数量，实现与 NoSQL 相当的线性扩展。

不过，由于数据基本上都是保存在内存中的，因此其容纳的数据总量会受到服务器安装的内存容量的限制。此外，虽然它具备复制和快照功能，但由硬件故障造成数据丢失的危险性，感觉比传统的数据库要高一些。而且，在数据库结构、构成集群的节点数量等系统结构的灵活性方面，和 NoSQL 等相比依然稍逊一筹，因此必须从一开始就对数据库结构进行精确的设计。

虽然性能很高，但却很难掌控，从这个意义上来看，VoltDB 可以说是数据库中的“方程式赛车”了。VoltDB 才刚刚诞生不久，今后也有进行诸多改善的计划。据说在这里提出的数据库结构和集群结构缺乏灵活性这一点，也将得到改善，在操作接口上也考虑支持 JSON 等等。VoltDB 可以说是今后值得期待的一种数据库。

5.5 memcached 和它的伙伴们

在程序的实现中，经常会忽略程序的运行时间。即便采用类似的实现方法，有时候运行速度也会相差很多。大多数情况下，这一速度上的差异是由数据访问速度的差异所导致的。

在程序中，虽然数据访问所消耗的时间看上去差不多，但实际上却有很大的差别。这是因为，数据访问所需要的时间，与数据存放的位置有很大关系。例如，内存中的数据与硬盘上的数据，其访问所需的时间可以相差数百万倍之多。

以机械旋转方式工作的硬盘，从驱动磁头到将盘片旋转到适当的扇区，需要几毫秒的时间，从 CPU 的速度来看这些时间都是需要等待的，而对内存的访问仅仅需要几纳秒的时间。相比之下，硬盘简直就像停止不动一样。此外，和位于外部的内存相比，位于 CPU 内部的寄存器和高速缓存的访问速度又能快上几倍。

用于高速访问的缓存

话虽如此，但内存的访问速度再快，要准备和硬盘同等容量的内存，将所有数据都保存在内存中，目前来看还是不现实的[①]。

所幸的是，对数据的访问具备“局部性”的特点。也就是说，一个操作中所访问的数据大多是可以限定范围的。即便数据的量很大，但大多数的操作都是仅仅对一部分数据进行频繁的访问，而几乎不会去碰其余的数据。

既然如此，如果将这些频繁访问的数据复制到一个可以高速访问的地方，平时就在那个地方进行操作的话，就有可能为性能带来大幅度的改善。像这样“能够高速访问的数据存放地点”，被称为缓存（cache）。“缓存”这个词的英文 cache，和“现金”的英文 cash 发音相同，但拼写方法是不同的。cache 一词来自法语，原来是“储藏地”、“仓库”的意思。最近的 CPU 中为了

① 不过，现在的服务器所配置的内存容量，已经超过了过去主流硬盘的容量。此外，比硬盘访问速度快几倍的闪存式 SSD（Solid State Drive，固态硬盘）也已经开始普及。说不定在不久的将来，现在的“常识”就会被颠覆。（原书注）

高速访问数据和指令，都配备了一定的高速缓存，但缓存一词本身，应该是泛指所有用于加速数据访问的手段。

提到缓存，往往会包含以下含义：

- ❑ 可以高速访问。
- ❑ 以改善性能为目的。
- ❑ 仅用于临时存放数据，当空间已满时可以任意丢弃多出来的数据。
- ❑ 数据是否存放在缓存中，不会产生除性能以外的其他影响。

数据库的职责是用来永久存储数据，不可能将数据任意丢弃。缓存则不同，它具有较大的随意性。

memcached

在提供缓存功能的软件中，比较流行的是 memcached。memcached 是由美国 Danga Interactive 公司在 Brad Fitzpatrick（1980— ）的带领下开发的一种“内存型键 – 值存储”软件，主要面向 Web 应用程序，对数据库查询的结果进行缓存处理。

当对数据库进行查询时，数据库服务器会执行下列操作：①解析 SQL 语句；②访问数据；③提取数据；④（根据需要）对数据进行更新等操作。考虑到数据库中的数据大多存放在磁盘上（当然，数据库服务器本身也有一定的缓存机制），这样的操作开销是非常大的。

另一方面，近年来，随着服务器及内存价格的下降，相比购买昂贵的高性能服务器来说，将查询结果缓存在内存中就可以以低成本实现高性能。因此，memcached 并不是一个真正意义上的键 – 值存储数据库，而是主要着眼于以缓存的方式来改善数据访问的性能。

用于实现缓存功能的 memcached 具备以下特征：

- ❑ 以字符串为对象的键 – 值存储。
- ❑ 数据只保存在内存中，重启 memcached 服务器将导致数据丢失。
- ❑ 可以对数据设置有效期。
- ❑ 达到一定容量后将清除最少被访问的数据。
- ❑ 键的长度上限为 250B，值的长度上限为 1MB。

memcached 能够接受的命令如表 1 所示。

memcached 非常好用，应用也非常广泛。根据其主页上的介绍，表 2 所示的这些服务都使用了 memcached。这些都是非常著名的网站或公司，当然，在很多没有那么有名的网站中，

memcached 的使用也十分广泛。

表1 memcached命令

名　称	功　能
set	设置数据
add	插入新数据
replace	更新现有数据
append	在值之后添加数据
prepend	在值之前添加数据
get	获取数据
gets	获取数据（有唯一ID）
cas	更新数据（指定唯一ID）
delete	删除数据
incr	将数据视为数值并累加
decr	将数据视为数值并减少
stats	获取统计数据
flush_all	清空数据
version	版本信息
verbosity	设置日志级别
quit	结束连接

表2 使用memcached的服务及企业

服务名称或企业名称
Bebo，Craigslist，Digg，Flickr，LiveJournal，mixi，Typepad，Twitter，Wikipedia，Wordpress，Yellowbot，YouTube

示例程序

memcached 可以通过 C、C++、PHP、Java、Python、Ruby、Perl、Erlang、Lua 等语言来访问。此外，memcached 的通信协议是由简单的文本形式构成的，使用如 telnet 等方式也很容易进行访问，要开发新的客户端也非常容易。除了上述列举的语言之外，还有很多语言也提供了 memcached 客户端库。

下面我们来看看访问 memcached 客户端程序的示例吧（图 1）。有很多库都提供了用 Ruby 访问 memcached 的功能，在这里我们用的是一个叫做 memcache 的库。正如图 1 最下方的注释所写的，对于同时执行查询和更新操作的数据进行缓存时一定要注意，否则一不小心就容易造成缓存和数据库之间的数据不匹配。通过 memcache 库也可以很容易地实现事务机制。下面我们来重新实现一下 prepend 命令吧（图 2）。

```
require 'memcache'

# 连接memcached
MCache = Memcache.new(:server => "localhost:11211")

# 对userinfo进行带缓存查询
def userinfo(userid)
  # 使用"user:<userid>"为键来访问缓存
  result = MCache.get("user:" + userid)

  # 如果不存在于缓存中则返回nil
  unless result
    # 缓存中不存在,直接查询数据库
    result = DB.select("SELECT * FROM users WHERE userid = ?", userid)
    # 将返回结果存放在缓存中,以便下次从缓存中查询
    MCache.add("user:" + userid,  result)
  end
  result
end

# 如果对userinfo进行更新,需要同时更新缓存
```

图 1　Ruby 编写的 memcached 客户端

```
def mc_prepend(key, val)
  loop do
    # :设置:cas标志并获取唯一ID
    v = MCache.get(key, :cas => true)
    # 修改值并用cas命令设置
    # 唯一ID可通过memcache_cas获取
    v = MCache.cas(key, val + v, :cas => v.memcache_cas)
    # 设置失败则返回nil(循环)
    return v unless v.nil?
  end
end
```

图 2　memcached 事务示例

对 memcached 的不满

和其他大多数软件一样，随着使用的不断增加，memcached 也遇到了当初从未设想过的应用场景，从而也招来了很多不满。对 memcached 的不满主要有下列这些：

数据长度：对键和值的长度分别限制在 250B 和 1MB 过于严格。越是大的数据查询起来就要花很长的时间，从这个角度来说，对这样的数据进行缓存的需求反而比较高。

分布：一台服务器的内存容量是有限的，如果能将缓存分布到多台服务器上就可以增加总的数据容量。然而 memcached 并没有提供分布功能。

持久性：顾名思义，memcached 只是一个缓存，重启服务器数据就会丢失，且为了保持一定的缓存大小，还会自动舍弃旧数据。

不过，当对数据库进行访问来填充缓存的开销超过一定程度时，缓存的损失就要付出较高的代价。为了克服这一问题,(即便舍弃缓存这一最初的目的)便产生了对数据进行持久化的需求。

像最后的“持久性”这一点，虽然已经完全脱离了缓存的范畴，但随着其应用领域的扩展，还是会产生各种各样类似的要求。

在 memcached 的范围内，对这些问题，主要是采取在客户端方面进行努力来应对。例如，Ruby 的 memcache 库中，提供了一个叫做“SegmentedServer”的功能，通过将键所对应的值分别存放到多台服务器中来支持长度很大的值。此外，还可以根据由键计算出的某种散列值，在客户端级别上实现对分布式数据存储的支持。其算法如图 3 所示。

```
# 表示算法的简单Ruby代码
class DistMemcache
  def add(key, value)
    # 由键计算散列值
    hash = hash_func(key)
    # 根据散列值选择服务器
    server = @servers[hash % @servers.size]
    # 向选择的服务器发送命令
    server.add(key, value)
  end
  # 对其他的命令采用同样的方法
end
```

图 3　memcache 在客户端级别实现分布

对于持久性这一点，在客户端级别上恐怕是无能为力了。不过新版本的 memcached 中正在对数据存储功能进行抽象化，从而实现将数据以文件及数据库等形式进行保存，重启服务器也不会丢失。

memcached 替代服务器

刚才已经讲过，memcached 的协议是基于文本的，非常简单[①]，因此大多数键 - 值存储数据库都可以支持 memcached 协议。下面我们来介绍其中的几种数据库软件。

1. memcachedb

名字只差了一个字母，很容易搞混。这是一款用 Berkeley DB 来保存数据的 memcached 替代服务器。memcachedb 原本是由 memcached 改造开发而来，目的是为了应对 memcached 不支持数据持久性的问题。

① 为了提高效率，也提供了二进制协议。(原书注)

2. ROMA

我所参与的乐天技术研究所开发的键 - 值存储 ROMA，也支持通过 memcached 协议来进行访问。ROMA 是一种重视可扩展性的键 - 值存储，其数据库是由 P2P 方式的节点集合所构成的。数据在多个节点上保存副本，即便由于一些故障导致一个节点退出服务，也不会造成数据的丢失。memcached 只是在客户端一侧实现了分布，相对而言，ROMA 则是在服务器一侧实现了分布和冗余化。

ROMA 服务器是用 Ruby 进行开发的，采用了可插式（pluggable）架构。通过用 Ruby 编写插件，并配置到服务器上，就可以很容易地实现数据保存方法的选择、服务器端命令的追加等。

采用 Ruby 进行实现，大家可能会担心性能方面的问题。ROMA 是在乐天内部的各种场景下运用的，由于采用多个节点分担负荷，实际运用中并没有发生性能方面的问题。其实，在乐天这样大规模数据中心的应用中，发生硬件故障、访问量集中导致进程终止等问题几乎是家常便饭，因此比起性能来说，ROMA 更加重视实现实际运用中的稳定性和灵活性。

3. Flare

Flare是由GREE[①]的CTO藤本真树领导开发的一种memcached替代键–值存储。其特征如下：

- ❑ 使用 Tokyo Cabinet 实现数据持久性
- ❑ 将数据复制到多台服务器上实现数据分区
- ❑ 无需停止系统就可以添加服务器的动态重组机制
- ❑ 节点监控 + 故障转移（failover）
- ❑ 可支持大于 256B 的键和大于 1MB 的值

根据文档，GREE 使用的 Flare 由 12 个节点（6 个主节点、6 个从节点）构成，目前正在对超过 2000 万个键和 1GB 级别的数据，以峰值每秒 500 ～ 1000 次访问的频率进行试运行。在这样的条件下，系统基本上没有什么负担，而且在运行中频繁更换服务器，也没有对系统的运转产生任何问题。

4. Tokyo Tyrant

Tokyo Tyrant 是由平林干雄开发的一种网络型键 – 值存储。很多键 – 值存储都在使用的 DBM 库 Tokyo Cabinet 也是平林先生开发的。相对而言，Tokyo Tyrant 提供了通过网络进行访问的功能。Tokyo Tyrant 支持 memcached 协议及 HTTP 协议，从这个角度来看，也可以视为

① GREE 是日本的一家社交网站（http://gree.jp），与中国的“格力电器”无关。

memcached 的替代服务。

Tokyo Tyrant 虽然需要写入磁盘，但却能够实现与 memcached 同等的性能。

5. kumofs

kumofs 是由筑波大学的古桥贞之（现供职于美国 Treasure Date 公司）开发的一种键－值存储，实现了持久性和冗余故障容忍性。

用 C++ 编写的 kumofs，仅用 1 个节点就能够实现与 memcached 几乎同等的性能，还可以通过增加节点来进一步提高性能。此外，它还能够在不停止系统运行的情况下，进行节点增加、恢复等操作，从而对访问量的增加等情况作出灵活的应对。

另一种键－值存储 Redis

看了上面这些例子，我们可以总结一下，大家对 memcached 的不满主要体现在以下几个方面：

① 缺乏持久性
② 不支持分布
③ 数据长度有限制

于是，为了解决这些不满，就出现了各种各样的替代服务器软件。然而，这里希望大家不要误会，这些不满决不等于是 memcached 的缺点。顾名思义，memcached 原本是作为数据缓存服务器诞生的，因此，缺乏持久性、不支持分布，以及对数据长度的限制，都是作者原本的意图。可以说，之所以会出现这些不满，都是试图将 memcached 用在超出原本目的的场景中所导致的结果。这些不满的存在表明，大多数的使用案例中，用户所需要的不仅仅是一个缓存数据库，而是一个支持持久性和分布式计算的真正意义上的键－值存储数据库。而之所以选择了原本并不适合这一目的的 memcached，无非是被 memcached 的高速性所吸引的结果。

也就是说，对同时满足：

① 高速
② 支持分布
③ 持久性

这些特性的键－值存储的需求，是真实存在的。而且，如果不但支持单纯的字符串键值对，而是能够以更加丰富的数据结构作为值来使用的话，大家也一定是非常喜欢的。能满足上述这些需求的，就是 Redis。Redis 是意大利程序员 Salvatore Sanfilippo 开发的一种内存型键－值存储，其特征如下：

内存型：数据库的操作基本都在内存中进行，速度非常快。

支持永久化：上面提到数据库的操作是在内存中进行的，但是它提供了异步输出文件的功能。发生故障时，最后输出的文件之后的变更会丢失。虽然并不具备严格的可靠性，但却可以避免数据的完全丢失。

支持分布：现行版本和 memcached 一样支持在客户端一侧实现对分布的支持。Redis Cluster 分布层的开发还在计划中。具备服务器端复制功能。

除字符串之外的数据结构：除了字符串，Redis 还支持列表（list，字符串数组）、集（set，不包含重复数据的集合）、有序集（sorted set，值经过排序的集合）和散列表（hash，键－值组合）。在 memcached 中必须强制转换为字符串才能存放的数据结构，在 Redis 中可以直接存放。

高速：全面使用 C 语言编写的 Redis 速度非常快。根据测试数据，在 Xeon 2.5GHz 的 Linux 计算机上，每秒能够处理超过 5 万个请求。在另一个测试中，对单节点采用 60 个线程分别产生 10000 个请求的调用，Redis 实现的每秒请求数量达到了 memcached 的两倍。而同样一个测试中，memcached 的成绩几乎是 MySQL 的 10 倍，从这个角度来看，Redis 的性能着实令人惊叹。

原子性：由于 Redis 内部是采用单线程实现的，因此各命令都具备原子性。像用 incr 命令对值进行累加而干扰其他请求的执行这样的问题是不会发生的。

不兼容 memcached 协议：Redis 拥有自己的数据结构，功能也比较丰富，因此没有采用 memcached 协议。访问 Redis 需要借助客户端库，但 Redis 协议也是基于文本的简单协议（实际上和 memcached 协议很相似），因此无论各种语言都很容易支持，包括 Ruby 在内，很多语言都提供了用于访问 Redis 的库的功能（表 3），数量上不输给 memcached。

Redis 的主页上列出了一些实际采用 Redis 的企业名单，如表 4 所示。其中，Engine Yard 和 GitHub 都是 Ruby 开发者耳熟能详的公司。

表3　提供Redis访问的语言

语言名称
C，C#，Clojure，Haskell，Io，Erlang，Java，JavaScript，Go，Perl，Lua，PHP，Python，Ruby，Scala，Tcl

表4　采用Redis的企业

企业名称
Boxcar，craigslist，Dark Curse，Engine Yard，GitHub，guardian，LLOOGG，OKNOtizie，RubyMinds，Superfeedr，Vidiowiki，Virgilio Film，Wish Internet Consulting，Zoombu

Redis 的数据类型

之前已经讲过，Redis 中的值，除了字符串以外，还支持列表、集、有序集、散列等数据结构。

字符串：将字符串作为值的操作和 memcached 几乎是一样的。不知道为什么，Redis 中没有提供在值之前添加字符串的 prepend 命令，也许是因为很少使用吧。实在需要这个功能的话，也可以通过事务来实现。

列表：相当于字符串数组。Redis 不支持如数组的数组这样的嵌套数据结构，因此数组的元素仅限于字符串。

列表是可以对左右两端进行 PUSH（添加一个元素）和 POP（取出并删除一个元素）操作的，因此可以作为队列和栈来使用。

集：相当于不允许出现重复元素的集合。通过 Redis 的 SADD 命令添加元素，用 SREM 命令删除元素。由于没有定义元素的顺序，因此使用取出一个元素的 SPOP 命令不知道会取出集中的哪个元素。

集合之间也可以进行运算，例如 SINTER 可以得到两个集中都包含的元素（交集），SUNION 则可以得到两个集中属于任意一个集的元素（合集）。此外，还可以用 SINTERSTORE/SUNIONSTORE 命令将运算结果保存到另一个键中。

有序集：Redis 从 1.1 版开始提供的一种有序的集（sorted set，Redis 术语中称为 ZSET），在这种集中会将元素都视为数值并进行排序。说实话我不知道这样的数据结构应该用在哪里，也许在某些场合中用起来会很方便吧。

散列表：散列表就是通过键来查找值的一种表。在 Redis 中，散列表的键和值都限定为字符串。

将上述特征总结一下，与只能存放字符串键值对的 memcached 相比，Redis 是一种高速、拥有丰富的数据结构，且支持异步快照功能的键 - 值存储。不过，Redis 也并非万能的数据库，（至少截至到目前来说）还不具备服务器端 Sharding（分布）和动态重组功能，也没有实现非常高的可靠性。

在数据不要求有很高的可靠性（也就是说，万一丢掉一些数据也不会产生严重后果）的情况下，Redis 就是一个非常理想的数据库。从 memcached 的使用实例来看，这样的领域还是非常广阔的。

Redis 的命令与示例

Redis 的命令如表 5 所示。虽然形式非常类似，但数量却远远多于 memcached 命令。

表5 Redis命令一览

命 令	概 要
连接管理	
QUIT	结束连接
AUTH	简单认证（可选）
数据库操作	
DBSIZE	当前 DB 中的键数量
SELECT index	数据库选择
MOVE key index	将键移动到 index 的 DB
FLUSHDB	删除当前 DB 中的全部键
FLUSHALL	删除全部 DB 中的全部键
适用于所有值类型的命令	
EXISTS key	检查 key 是否存在
DEL key	删除 key
TYPE key	key 的类型
KEYS pattern	列出与 pattern 相匹配的键
RANDOMKEY	随机获取一个键
RENAME old new	重命名键（new 已存在则舍弃）
RENAMENX old new	重命名键（new 已存在则忽略）
EXPIRE key sec	设置有效期
TTL key	剩余有效期
适用于字符串值的命令	
SET key val	将 val（字符串）赋值给 key
GET key	获取 key 相对应的值
GETSET key val	将 val 赋值给 key 并获取更新前的原始值
MGET key1 key2 ... keyN	获取多个 key 相对应的值
SETNX key val	key 不存在时更新 val
SETEX key time val	带有效期的 SET
MSET key1 value1 key2 value2... keyN valueN	对多个 key 和 val 进行更新（原子操作）
MSETNX key1 value1 key2 value2 ... keyN valueN	对多个 key 和 val 进行更新（原子操作，其中任何 key 都不能是事先存在的）
INCR key	对值加 1
INCRBY key int	对值加 int
DECR key	对值减 1
DECRBY key int	对值减 int

（续）

命　　令	概　　要
APPEND key val	在值末尾追加val
SUBSTR key start end	获取值字符串的一部分
适用于列表值的命令	
RPUSH key val	向列表末尾追加val
LPUSH key val	向列表开头追加val
LLEN key	列表长度
LRANGE key start end	列表中指定范围内的元素
LTRIM key start end	将列表按指定范围切割
LINDEX key index	指定位置的元素
LSET key index val	向指定位置写入val
LREM key count val	从列表开头删除count个val（0为删除全部，负数为从末尾开始删除）
LPOP key	获取并删除列表开头的元素
RPOP key	获取并删除列表末尾的元素
BLPOP key1 key2 ... keyN timeout	带超时的LPOP
BRPOP key1 key2 ... keyN timeout	带超时的RPOP
RPOPLPUSH key1 key2	从key1列表中RPOP，然后LPUSH到key2列表中
适用于集值的命令	
SADD key member	向集添加元素
SREM key member	从集中删除元素
SPOP key	从集中随机删除一个元素
SMOVE key1 key2 member	将member从key1集移动到key2集（原子操作）
SCARD key	集的元素数
SISMEMBER key member	key集中是否包含元素member
SINTER key1 key2 ... keyN	属于所有指定集的元素
SINTERSTORE dstkey key1 key2 ... keyN	将属于所有指定集的元素的集赋值给dstkey
SUNION key1 key2 ... keyN	属于任一指定集的元素
SUNIONSTORE dstkey key1 key2 ... keyN	将属于任一指定集的元素的集赋值给dstkey
SDIFF key1 key2 ... keyN	key1与Key2及其之后的集之间的差异元素
SDIFFSTORE dstkey key1 key2 ... keyN	将key1与Key2及其之后的集之间的差异元素赋值给dstkey
SMEMBERS key	集中所有的元素
SRANDMEMBER key	随机获取集中一个元素
适用于有序集（ZSET）的命令	
ZADD key score member	添加成员（已经存在则更新score）
ZREM key member	删除member
ZINCRBY key inc member	将member的得分增加inc
ZRANK key member	member的排位
ZREVRANK key member	member的排位（倒序）

（续）

命　令	概　要
ZRANGE key start end	获取范围内的元素
ZREVRANGE key start end	获取范围内的元素（倒序）
ZRANGEBYSCORE key min max	获取得分为min到max之间的元素
ZCARD key	有序集的元素数
ZSCORE key member	member的得分
ZREMRANGEBYRANK key min max	获取排位为min到max之间的元素
ZREMRANGEBYSCORE key min max	删除得分为min到max之间的元素
ZINTERSTORE dstkey N key1 ... keyN	将指定有序集的交集赋值给dstkey
ZUNIONSTORE dstkey N key1 ... keyN	将指定有序集的合集赋值给dstkey
适用于散列表的命令	
HSET key field val	对key指定的散列表的field设置val
HGET key field	获取key指定的散列表的field
HMGET key field1 ... fieldN	获取多个field对应的值
HMSET key field1 value1 ... fieldN valueN	设置多个field（原子操作）
HINCRBY key field int	将field的值增加int
HEXISTS key field	判断散列表中是否包含field
HDEL key field	删除field
HLEN key	散列表的field数
HKEYS key	获取散列表的所有field
HVALS key	获取散列表的所有value
HGETALL key	获取散列表的所有field和value
排　序	
SORT key	对列表、集、有序集进行排序
事　务	
MULTI/EXEC/DISCARD/WATCH/UNWATCH	Redis的原子性事务
Publish/Subscribe	
SUBSCRIBE/UNSUBSCRIBE/PUBLISH	Redis Publish/Subscribe通信
持久性控制命令	
SAVE	同步保存
BGSAVE	异步保存
LASTSAVE	最后保存时间
SHUTDOWN	同步保存后停止服务器
BGREWRITEAOF	替换日志文件
远程服务器控制命令	
INFO	服务器信息
MONITOR	请求转储（调试用）
SLAVEOF	复制的设置
CONFIG	Redis设置（可变更）

我们将图 1 的 memcached 客户端改造成了 Redis 客户端（图 4）。Ruby 的 memcache 库可以将对象自动转换成字符串，不过 Redis 库却不会进行这样的自动转换。因此，我们需要用 Marshal 显式地转换成字符串。除了这一点和 memcached 客户端有区别以外，其他方面基本上是相同的。在这里我们只使用了字符串，不过用 Redis 的散列来表达 userinfo 可能也挺有意思的。

```
require 'redis'

# 连接Redis
RD = Redis.new(:host => "localhost", :port => 6379)

# 对userinfo进行带缓存查询
def userinfo(userid)
  # 使用“user:<userid>”为键来访问缓存
  result = RD.get("user:" + userid)

  # 如果不存在于缓存中则返回nil
  unless result
    # 缓存中不存在,直接查询数据库
    result = DB.select("SELECT * FROM users WHERE userid = ?", userid)
    # 将返回结果存放在缓存中,以便下次从缓存中查询
    RD.set("user:" + userid, Marshal.dump(result))
  else
    # 将缓存还原成对象
    # redis库不会进行自动字符串转换
    result = Marshal.load(result)
  end
  result
end
```

图 4　Ruby 编写的 Redis 客户端

Redis 中没有像 memcached 的 prepend 这样的命令，要进行这样的原子性操作就需要用到事务。memcached 和 Redis 在事务的结构方面是有很大差异的。Redis 的事务是由包裹在 multi 命令和 exec 命令之间的部分构成的。当遇到 multi 命令时，其后面的所有命令都只进行参数检查，然后记录到日志中，随后当遇到 exec 命令时，再一次性（原子性）地执行所有的记录下来的命令集。如果需要像 prepend 这样对现有 key 进行变更，则需要事先用 watch 命令指定要变更的 key。

图 5 是一个通过 Redis 的事务来实现 prepend 的例子。不过，在编写这个程序的时候，Redis 的事务功能还处于开发阶段，因此无法使用 WATCH 命令。

```
def mc_prepend(key, val)
  loop do
    RD.watch(key)
    v = RD.get(key) + val
    RD.multi
    RD.set(key, v)
    unless RD.exec.nil?
      return v
    end
  end
end
```

图 5　Redis 事务示例

如果没有事务功能的话，可能很多人会感觉非常别扭。但是，在采用 Redis 的场景中，并不一定需要事务功能，因此我觉得开发和完善这个功能的优先级并不高。大家可以回想一下，在 Web 应用程序中广泛使用的 MySQL 数据库，也是在曾经很长一段时间内都不支持事务的。

小结

memcached 以及对其“不满”的应对，似乎都是云计算网络环境改变了对软件的要求所导致的结果。环境的变化，必然会加速软件的进化。

“支撑大数据的数据存储技术”后记

历史上，对于数据存储的重大革新，可以说非 RDB（关系型数据库）莫属了。具备关系代数理论背景的 RDB，虽然在 1970 年诞生之初遭到了无数批评，称其毫无用武之地，然而现在 RDB 却几乎已经成为了数据库的代名词。

不过，在进入云计算时代之后，除 RDB 以外的其他方案开始受到越来越多的关注，本章中也对其中的 MongoDB、memcached 和 Redis 等进行了介绍。由于这些数据库系统并非采用 SQL 的 RDB，因此经常被称为 NoSQL。它们之所以备受关注，应该说是因为在大量节点构成的云计算系统中，数据库服务器逐渐成为了整个系统的瓶颈。而且，大多以通用数据库服务器形式提供的 RDB，出于各种各样的原因，很难解决这一瓶颈问题。

当然，要解决这一问题，也可以采用复制（对多个数据库进行同步，并将请求分配到多个数据库服务器上）、分割（按照一定的标准将数据库分割成多个，例如将编号为偶数和奇数的会员分别保存到不同的数据库中）等技术，但这些技术都需要在客户端一侧提供一定的支持，而从结果来说，有很多场景只是需要通过键来获取值这样简单的操作，并不一定要动用 RDB。

在这样的背景下，就出现了只管理键值对并将数据完全保存在内存中的缓存系统 memcached。同时，由于缓存数据丢失后还是需要访问数据库，与其产生这样的开销还不如自己来管理文件写入操作，于是就诞生了 ROMA、Flare 等软件。

除此之外，随着对系统灵活性的需求不断提高，固定的数据库结构（schema）已经无法应对各种变化，于是便出现了追求数据库结构灵活性的 MongoDB 和 Redis。另一方面，RDB 也认识到其在速度和灵活性方面的问题，同时为了解决这些问题而进行着持续的进化。本章中介绍的 VoltDB 正是其中的一种尝试。

在这样的对峙中，数据存储的基础，即存储架构本身也在不断发生变化。速度缓慢的硬盘（HDD）正在被淘汰，数据存储逐步过渡到采用以闪存为基础的固态硬盘（SSD），同时，MRAM、FeRAM 等下一代内存也开始崭露头角，据说可以实现和现有 DRAM 同等的速度、能够与闪存相媲美的容量，而且还能实现永久性数据存储（断电后数据不会丢失）。这样的话，数据库这一概念就有可能会从根本上被颠覆。

在下一代内存得到广泛运用的时代，曾经像 Smalltalk、Lisp 那样将内存空间直接保存下来的模型，说不定会东山再起。

第6章 多核时代的编程

6.1 摩尔定律

关于摩尔定律，本书中已经提到了很多次。摩尔定律是由美国英特尔公司的戈登 · 摩尔（Gordon Moore）提出的，指的是“集成电路中的晶体管数量大约每两年翻一倍”。下面我们就摩尔定律进行一些更深入的思考。

实际上，在 1965 年的原始论文中写的是“每年翻一倍”，在 10 年后的 1975 年发表的论文中又改成了“每两年翻一倍”。在过去的 40 年中，CPU 的性能大约是每一年半翻一倍，因此有很多人以为摩尔定律的内容本来是“每 18 个月翻一倍”。

其实，在几年前对此进行考证之前，我也是这么以为的。然而，似乎没有证据表明戈登 · 摩尔提出过“18 个月”这个说法。但英特尔公司的 David House 曾经在发言中提到过“LSI（大规模集成电路）的性能每 18 个月翻一倍”，因此 18 个月一说应该是起源于他。

虽然摩尔定律也叫定律，但它并非像物理定律那样严格，而只是一种经验法则、技术趋势或者说是目标。然而，令人惊讶的是，从 1965 年起至今，这一定律一直成立，并对社会产生了巨大的影响。

呈几何级数增长

“两年变为原来的两倍”，就是说 4 年 4 倍、6 年 8 倍、$2n$ 年 2 的 n 次方倍这样的增长速度。像这样“n 年变为 K 的 m 次方倍”的增长称为几何级数增长。

对于我们来说，摩尔定律的结果已经司空见惯了，也许一下子很难体会到其惊人的程度。下面我们通过一个故事，来看一看这种增长的速度是何等令人震惊。

很久很久以前，在某个地方有一位围棋大师，他的围棋水平天下无双，于是领主说：“你想要什么我就可以赏给你什么。”大师说：“我的愿望很简单，只要按照棋盘的格子数，每天给我一定数量的米就可以了。第一天一粒米，第二天两粒米，每天都比前一天的粒数翻倍。”

“什么嘛，从一粒米开始吗？”领主笑道，“你可真是无欲无求啊。好，明天就开始吧。”围棋的棋盘有 19×19 个格子，也就是说领主要在 361 天中每天赏给大师相应的米。第一天给 1 粒，

第二天是两粒，然后是 4 粒、8 粒、16 粒、32 粒。一开始大家都觉得："也就这么点米嘛。"但过了几天之后情况就发生了变化。两周还没到，赏赐的米粒一碗已经装不下了，要用更大的盆子才能装下，这时，有一位家臣发现情况不妙。

"主公，大事不好！""怎么了？""就是赏给大师的那些米，我算了一下，这个米的数量可不得了，最后一天，也就是第 361 天，要赏给他的米居然有 2348542582773833227889480596789337027375682548908319870707290971532209025114608443463698998384768703031934976 粒。这么多米，别说我们这座城，就是全世界的米都加起来也不够啊！[①]""天呐！"无奈，领主只能把大师叫来，请他换一个愿望。

看了上面这个故事，我想大家应该明白几何级数增长会达到一个多么惊人的数字了。而在半导体业界，这样的增长已经持续了 40 多年。大量技术人员不懈努力才将这样的奇迹变成现实，这是一项多么了不起的成就啊。

摩尔定律的内涵

半导体的制造使用的是一种类似印刷的技术。简单来说，是在被称为"晶圆"（wafer）的圆形单晶硅薄片上涂一层感光树脂（光刻胶），然后将电路的影像照射到晶圆上。其中被光照射到并感光的部分树脂会保留下来，其余的部分会露出硅层[②]。接下来，对露出的硅的部分进行加工，就可以制作成晶体管等元件。摩尔定律的本质，即如何才能在晶圆上蚀刻出更细微的电路，是对技术人员的一项巨大的挑战。

技术人员可不是为了自我满足才不断开发这种细微加工工艺的。电路的制程缩小一半，就意味着同样的电路在硅晶圆上所占用的面积可以缩小到原来的 1/4。也就是说，在电路设计不变的情况下，用相同面积的硅晶圆就可以制造出 4 倍数量的集成电路，材料成本也可以缩减到原来的 1/4。

缩减制程的好处还不仅如此。构成 CPU 的 MOS（Metal-Oxide Semiconductor，金属氧化物半导体）晶体管，当制程缩减到原来的 1/2 时，就可以实现 2 倍的开关速度和 1/4 的耗电量。这一性质是由 IBM 的 Robert Dennard[③]发现的，因此被命名为 Dennard Scaling。

---

① 实际上这个数字已经超过了宇宙中存在的所有粒子的数量。（原书注）

② 光刻胶有两种：一种在感光之后可以被显影剂溶解，即正光刻胶；另一种在感光之后不会被显影剂溶解，即负光刻胶。这里提到的是负光刻胶。

③ Robert Dennard（1932— ）是美国电子工程师，发明家，他于 1968 年发明了目前广为使用的 DRAM 内存。

综上所述，如果制程缩减一半，就意味着可以用同样的材料，制造出 4 倍数量、2 倍速度、1/4 耗电量的集成电路，这些好处相当诱人，40 多年来摩尔定律能够一直成立，其理由也正在于此。缩减制程所带来的好处如此之大，足以吸引企业投入巨额的研发经费，甚至出资建设新的半导体制造工厂也在所不惜。

摩尔定律的结果

可以说，最近的计算机进化和普及，基本上都是托了摩尔定律的福。半导体技术的发展将摩尔定律变为可能，也推动了计算机性能的提高、存储媒体等容量的增加，以及价格难以置信般的下降。

例如，现在一般的个人电脑价格都不超过 10 万日元（约合人民币 8000 元），但其处理性能已经超过了 30 年前的超级计算机。而且，当时的超级计算机光租金就要超过每月 1 亿日元（约合人民币 800 万元），从这一点上来说，变化可谓是天翻地覆的。

30 年前（1980 年左右）的个人电脑，我能想到的就是 NEC（日本电气）的 PC-8001（1979 年发售），和现在的电脑对比一下，我们可以看到一些非常有趣的变化（表 1）。

即使不考虑这 30 年间物价水平的变化，这一差距也可谓是压倒性的。而且，现在的笔记本电脑还配备了液晶显示屏、大容量硬盘和网络接口等设备，而 30 年前最低配置的 PC-8001 除了主机之外，甚至都没有配备显示屏和软驱，这一点也很值得关注。

表1　30年间个人计算机的变化

	PC-8001（NEC）	ThinkPad X201（Lenovo）	比　值
价格	16万8000日元（约合人民币1万3000元）	13万4820日元（约合人民币1万元）	0.8倍
CPU	Z80兼容4MHz	Intel Core i5 2.66GHz	655倍[①]
存储容量			
RAM	32KB	4GB	125万倍
ROM	24KB	---	---
外部存储器	软盘320KB	硬盘500GB	156万倍

① PC-8001 由于中断等待的原因，其有效时钟频率只有 2.3MHz 左右。而 X201 的 Core i5 由于具备睿频加速（Turbo Boost）功能，最高时钟频率可达到 3.2GHz。因此两者性能比值最高为 1391 倍。（原书注）

摩尔定律所带来的可能性

不过，摩尔定律所指的只是集成电路中晶体管数量呈几何级数增长这一趋势，而计算机性能的提高、价格的下降，以及其他各种变化，都是晶体管数量增长所带来的结果。

让我们来思考一下，通过工艺的精细化而不断增加的晶体管，是如何实现上述这些结果的呢？最容易理解的应该就是价格了。单位面积中晶体管数量的增加，同时也就意味着晶体管的单价呈几何级数下降。当然，工艺的精细化必然需要技术革新的成本，但这种成本完全可以被量产效应所抵消。

工艺的精细化，意味着制造相同设计的集成电路所需的成本越来越低。即便算上后面所提到的为提升性能而消费的晶体管，其数量的增长也是绰绰有余的。也就是说，只要工艺的精细化能够得以不断地推进，成本方面就不会存在什么问题。不仅是 CPU，电脑本身就是电子元件的集合。像这样由工艺改善带来的成本下降，就是上面所提到的 30 年来个人电脑在价格方面进化的原动力。

精细化所带来的好处并不仅仅是降低成本。由于前面提到的 Dennard Scaling 效应，晶体管的开关速度也得以实现飞跃性的提升。相应地，CPU 的工作时钟频率也不断提高。30 年前 CPU 的工作时钟频率还只有几 Mhz，而现在却已经有几 GHz 了，实际提高了差不多 1000 倍。

由于构成 CPU 的晶体管数量大幅增加，通过充分利用这些晶体管来提高性能，也为 CPU 的高速化做出了贡献。现代的 CPU 中搭载了很多高速化方面的技术，例如将命令处理分割成多段并行执行的流水线处理（pipeline）；不直接执行机器语言，而是先转换为更加细化的内部指令的微指令编码（micro-operation decoding）；先判断指令之间的依赖关系，对没有依赖关系的指令改变执行顺序进行乱序执行（out-of-order execution）；条件分支时不等待条件判断结果，而是先继续尝试执行投机执行（speculative execution）等。

在现代 CPU 的内部，都配备了专用的高速缓存，通过高速缓存可以在访问内存时缩短等待时间。从 CPU 的运行速度来看，通过外部总线连接的主内存访问起来非常缓慢。仅仅是等待数据从内存传输过来的这段时间，CPU 就可以执行数百条指令。

还好，对内存的访问存在局部性特点，也就是相同的数据具有被反复访问的倾向，因此只要将读取过的数据存放在位于 CPU 内部的快速存储器中，就可以避免反复访问内存所带来的巨大开销。这种方法就是高速缓存。缓存英文写作 cache，原本是法语“隐藏”的意思，大概指的是将内存中的数据贮藏起来的意思吧。

不过，CPU 内部配备的高速缓存容量是有限的，因此也有不少 CPU 配备了作为第二梯队的

二级缓存。相比能够从 CPU 直接访问的高速、高价、低容量的一级缓存来说，二级缓存虽然速度较慢（但仍然比内存的访问速度高很多），但容量很大。还有一些 CPU 甚至配备了作为第三梯队的三级缓存。如果没有高速缓存的话，每次访问内存的时候，CPU 都必须等待能够执行数百条指令的漫长时间。

最近的电脑中已经逐渐普及的多核和超线程（Hyper Threading）等技术，都是利用晶体管数量来提高运算性能的尝试。

为了提高性能

接下来，我们就来具体看一看，那些增加的晶体管到底是如何被用来提高 CPU 性能的。

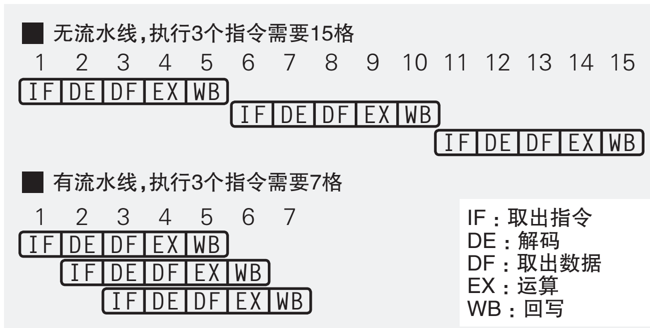


图 1　CPU 的流水线处理

CPU 在运行软件的时候，看起来似乎是逐一执行指令的，但其实构成 CPU 的硬件（电路）是能够同时执行多个操作的。将指令执行的操作进行分割，通过流水作业的方式缩短每一个单独步骤的处理时间，从而提升指令整体的执行速度，这种流水线处理就是一种提高性能的基本技术（图 1）。

典型的处理步骤包括：①取出指令（fetch）；②指令解码（decode）；③取出运算数据（data fetch）；④运算；⑤输出运算结果（write-back）等。

我们可以看出，将操作划分得越细，每一级的处理时间也会相应缩短，从而提升指令执行的吞吐量。出于这样的考虑，现代的 CPU 中流水线都被进一步细分，例如在 Pentium4 中被细分为 31 级（英特尔最新的 Core 架构是采用 14 级的设计）。

不过，流水线处理也并非十全十美。当流水作业顺利执行的时候是没什么问题的，一旦流水线上发生一个问题，就会接连引发一连串的问题。要想让流水线处理顺利进行，需要让各步骤都以相同的步伐并肩前进，而这一条件并非总能得到满足。

我们来看一个 CPU 加法指令的例子。x86 的加法指令是：

```
ADD a b
```

这条指令的意思是将 a 和 b 相加，并将结果保存在 a 中。a 和 b 可以是寄存器，也可以是内存地址，但对于 CPU 来说，访问寄存器和访问内存所需要的时间是天壤之别的。如果需要对内存进行访问，则在执行取出数据这一步的时间内，整个流水线就需要等待几百个时钟周期，这样一来流水线化对指令执行速度带来的那一点提升也就被抵消了。

像这样流水线发生停顿的问题被称为气泡（bubble / pipeline stall）。产生气泡的原因有很多种，需要针对不同的原因采取不同的对策。

上述这样由于内存访问速度缓慢导致的流水线停顿问题，被称为“数据冒险”（data hazard），针对这种问题的对策，就是我们刚刚提到过的“高速缓存”。高速缓存，实际上是消耗一定数量的晶体管用作 CPU 内部高速存储空间，从而提升速度的一种技术。

然而，高速缓存也不是万能的。即使晶体管数量大幅增长，其数量也不是无限的，因此高速缓存在容量上是有限制的。而且，缓存的基本工作方式是“将读取过一次的数据保存下来，使下次无需重新读取”，因此对于从未读取过的数据，依然还是要花费几百个时钟周期去访问位于 CPU 外部的内存才行。

还有其他一些原因会产生气泡，例如由于 CPU 内部电路等不足导致的资源冒险（resource hazard）；由于条件分支导致的分支冒险（branch hazard）等。资源冒险可以通过增设内部电路来进行一定程度的缓解。

这里需要讲解一下分支冒险。在 CPU 内部遇到条件分支指令时，需要根据之前命令的执行结果，来判断接下来要执行的指令的位置。不过，指令的执行结果要等到该命令的 WB（回写）步骤完成之后才能知晓，因此流水线的流向就会变得不明确（图 2）。

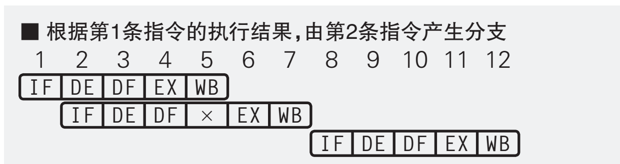


图 2　分支冒险

在图 2 中，首先执行第 1 条指令，与之并行执行第 2 条指令的取出操作（到第 4 个周期）。然而，第 1 条指令执行完毕之前，无法执行第 2 条指令的分支（第 5 个周期），这算是一种资源冒险吧。第 1 条指令的执行完全结束之后，才可以轮到第 2 条指令，而第 2 条指令的回写操作完成之后，才能够确定第 3 条指令位于哪个位置，也就是说，这时能够执行第 3 条指令的取出操作。

分支冒险可没那么容易解决。分支预测是其中的一种方案。分支预测是利用分支指令跳转目标上的偏向性，事先对跳转的目标进行猜测，并执行相应的取出指令操作。

图 3 是分支预测的执行示意图。到第 2 个指令为止的部分，和图 2 是相同的，但为了避免产生气泡，这里对分支后的指令进行预测并开始取出指令的操作。当预测正确时，整个执行过程需要 9 个周期，和无分支的情况相比只增加了 2 个。当预测错误时，流水线会被清空并从头开始。只要猜中就赚到了，没猜中也只是和不进行预测的结果一样而已，因此整体的平均执行速度便得到了提升。

最近的 CPU 已经超越了分支预测，发展出更进一步的投机执行技术。所谓投机执行，就是对条件分支后的跳转目标进行预测后，不仅仅是执行取出命令的操作，还会进一步执行实际的运算操作。当然，当条件分支的预测错误时，需要取消刚才的执行，但当预测正确时，对性能的提升就可以比仅进行分支预测来得更加高效。

流水线是一种在垂直方向上对指令处理进行重叠来提升性能的技术，相对地，在水平方向上将指令进行重叠的技术称为超标量（superscalar）。也就是说，在没有相互依赖关系的前提下，多条指令可以同时执行。

例如，同时执行两条指令的超标量执行情况如图 4 所示。从理论上说，最好的情况下，执行 6 条指令只需要 7 个周期,这真的是了不起的加速效果。在图 3 的例子中是同时执行两条指令，但只要增加执行单元，就可以将理论极限提高到 3 倍甚至 4 倍。实际上，在最新的 CPU 中，可同时执行的指令数量大约为 5 条左右。

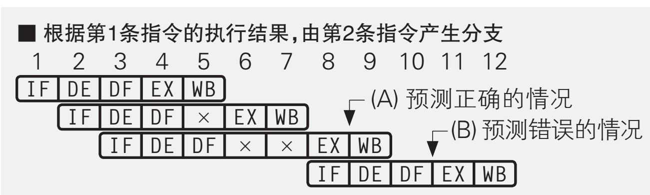


图 3　分支预测

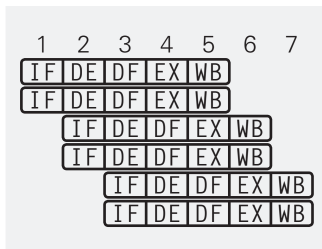


图 4　超标量执行

不过，事情总是没有这么简单。采用超标量架构的 CPU，实际能够同时执行的指令数量要远远低于理想的值，这是因为数据间的依赖关系妨碍了指令的同时执行。例如：

```
a = b + c
d = e + f
g = h + i
```

像上述这样的运算，各行运算之间没有相互依赖关系，最极端的情况下，即便打乱这些运算的顺序，结果也不会发生任何变化。像这样的情况，就能够发挥出超标量的最大性能。然而，如果是下面这样的话：

```
a = b + c
d = a + e
g = d + h
```

第 2 行的运算依赖第 1 行的结果，第 3 行的运算依赖第 2 行的结果。这就意味着第 1 行的运算得出结果之前，无法执行第 2 行的运算；第 2 行的运算得出结果之前，也无法执行第 3 行的运算，也就是说，无法实现同时执行。

于是，为了增加能够同时执行的指令数量，可以使用“乱序执行”技术。这个问题的本质在于，一个指令和用于计算它所依赖的结果的指令距离太近。正是由于相互依赖的指令距离太近，才导致 CPU 没有时间完成相应的准备工作。

那么，我们可以在不改变计算结果的范围内，改变指令的执行顺序，这就是乱序执行。乱序执行的英文 out of order 原本多指“故障”的意思，但这里“order”指的是顺序，也就是命令执行顺序与排列顺序不同的意思。

例如：

```
a = b + c
d = a + e（依赖第1行）
g = d + h（依赖第2行）
j = k + l
m = n + o
```

这样的运算，如果将顺序改为：

```
a = b + c
j = k + l
d = a + e（依赖第1行）
m = n + o
g = d + h（依赖第3行）
```

就可以填满空闲的执行单元，顺利的话，就能够稀释指令之间的相互依赖关系，从而提高执行效率。

为了充分利用流水线带来的好处，出现了一种叫做 RISC 的 CPU 架构。RISC 是 Reduced Instruction Set Computer（精简指令集）的缩写，它具备以下特征：

- ❑ 精简且高度对称的指令集。
- ❑ 指令长度完全相同（也有例外）。
- ❑ 和传统 CPU 相比寄存器数量更多。
- ❑ 运算的操作数只能为寄存器，内存中的数据需要显式地加载到寄存器中。

这样的特征所要达到的目的如下：

- ❑ 通过减少指令种类使电路设计简单化（高速化）。
- ❑ 通过统一指令的粒度使流水线更加容易维持。
- ❑ 根据依赖关系对指令的重新排序可通过编译器的优化来实现。

大约 20 年之前，RISC 架构是非常流行的，其中比较有名的有 MIPS 和 SPARC 等。现在，RISC 虽然没有被非常广泛的应用，但像智能手机中使用的 ARM 处理器就属于 RISC 架构，此外，PlayStation 3 等设备中采用的 CELL 芯片也是 RISC 架构的。

不过，RISC 的指令集与传统 CPU（与 RISC 相对的叫做 CISC：Complex Instruction Set Computer，复杂指令集）的指令集是完全不同的，它们之间完全不具备兼容性，这也成为了一个问题。过去的软件资产都无法充分利用，这不得不说是一个很大的障碍。

于是，最近的 x86 系 CPU 中，使用微指令转换技术，在保持传统 CISC 指令的同时，试图获得一些 RISC 的优势。这种技术就是在外部依然使用传统 x86 指令集的同时，在内部将 x86 指令转换为粒度更小的 RISC 型指令集来执行。一条 x86 指令会被转换成多条微指令。通过这种方式，在保持兼容 x86 指令集的同时，根据软件的实际情况，可以获得除依赖关系控制之外的 RISC 优势。但即便如此，要填充所有超标量的执行单元，还是十分困难的。

那么，为什么执行单元无法被有效填充呢？原因在于数据之间存在相互依赖关系。既然如此，那可以将没有依赖关系的多个执行同时进行吧？这也就是所谓的超线程（Hyper Threading）技术。超线程是英特尔公司的一个专有名词，这一技术的一般名称应该叫做 SMT（Simultaneous Multi-Threading，同时多线程），不过为了简便起见，这里统一使用超线程一词。

所谓超线程，就是通过同时处理多个取出并执行指令的控制流程，从而将没有相互依赖关系的运算同时送入运算器中，通过这一手段，可以提高超标量的利用效率。实际上，为了同时处理多个控制流程（线程），还需要增加相应的寄存器等资源。

超线程是对空闲运算器的一种有效利用，但并不是说可以按线程数量成比例地提高性能。根据英特尔公司发布的数据，超线程最多可提升 30% 左右的性能。不过，为了实现这 30% 的性能提升，晶体管数量仅仅增加了 5%。用 5% 的晶体管增加换取 30% 的性能提升，应该说是一笔划算的交易。

除了上述这些以外，还有其他一些提高性能的方法。例如在一块芯片中封装多个核心的多核（multi-core）技术。最近的操作系统中，多进程早已司空见惯，对多核的运用空间也愈发广阔。多核分为两种形式，即包含多个相同种类核心的同构多核（homogeneous multi-core），以及包含多个不同种类核心的异构多核（heterogeneous multi-core）。在异构多核中，除了通常的 CPU 以外，还可以包含 GPU（Graphic Processing Unit，图像处理单元）和视频编码核心等。此

外，包含数十个甚至数百个核心的芯片也正在研究，这被称为超多核（many-core）。

摩尔定律的极限

在过去40年里一直不断改变世界的摩尔定律，在今后是否能够继续有效下去，从目前的形势上看并不乐观。出于几个理由，芯片集成度的提高似乎已经接近了极限。

第一个极限就是导线宽度。随着半导体制造技术的进步，到2010年时，最小的制程已经达到32nm（纳米，1纳米为1米的10亿分之一）。刚才已经讲过，集成电路是采用一种类似印刷的技术来制造的，即用光照射模板，按照模板上的图案感光并在半导体上形成电路。问题是，电路已经变得过于精密，甚至比感光光源的波长还要小。目前采用的感光光源是紫外激光，而紫外激光的波长为96.5nm。

在森林里，阳光透过茂密的树叶在地面上投下的影子会变得模糊，无法分辨出一枚枚单独的树叶。同样，当图案比光的波长还小时，也会发生模糊而无法清晰感光的情况。为了能够印制出比光的波长更细小的电路，人们采用了各种各样的方法，例如在透镜和晶圆之间填充纯水来缩短光的波长等，但这个极限迟早会到来。下一步恐怕会使用波长更短的远紫外线或X射线。但波长太短的话，透镜也就无法使用了[①]，处理起来十分困难。或许可以用反射镜来替代透镜，但曝光机构会变得非常庞大，成本也会上升。

其次，即便这样真的能够形成更加细微的电路，还会发生另外一些问题。当电路变得非常精细时,就会发生一些超出经典物理而进入量子物理范畴的现象,其中一个例子就是“隧穿效应”。关于隧穿效应的详细知识在这里就省略了（因为我自己也不太明白），简单来说，即便是电流本该无法通过的绝缘体，在微观尺度上也会有少量电子能够穿透并产生微弱的电流。这样的电流被称为渗漏电流，现代CPU中有一半以上的电力都消耗在了渗漏电流上。

精密电路中还会产生发热问题。电流在电路中流过就会产生热量，而随着电路的精密化，其热密度（单位面积所产生的热量）也随之上升。现代CPU的热密度已经和电烤炉差不多了，如果不用风扇等进行冷却，恐怕真的可以用来煎蛋了。上面提到的渗漏电流也会转化为热量，因此它也是提升热密度的因素之一。

假设电路的精密化还保持和现在一样的速度，恐怕不久的将来就会看到这样的情形——按下开关的一瞬间整个电路就蒸发掉了（如果没有适当的冷却措施的话）。

① X射线很难使用透镜来聚焦。

最后,也是最大的一个难题,就是需求的饱和。最近的电脑 CPU 性能已经显得有些驻足不前。CPU 指标中最为人知的应该就是主频了。尽管每个 CPU 单位频率的性能有所差异，但过去一直快速增长的 CPU 主频，从几年前开始就在 2GHz 水平上止步不前，即便是高端 CPU 也是如此。而过去在 Pentium 4 时代还能够见到的 4GHz 级别主频的产品，今天已经销声匿迹了。

这是因为，像收发邮件、浏览网页、撰写文稿这些一般大众级别的应用所需要的电脑性能，用低端 CPU 就完全可以满足，主频竞争的降温与这一现状不无关系。

进一步说，过去人们一直习惯于认为 CPU 的性能是由主频决定的。而现在，在多核等技术的影响下，主频已经不是决定性能的唯一因素了。这也成为主频竞争的必然性日趋下降的一个原因。实际上，以高主频著称的 Pentium 4，其单位频率性能却不怎么高，可以说是被主频竞争所扭曲的一代 CPU 吧。

上面介绍的这些对摩尔定律所构成的障碍，依靠各种技术革新来克服它们应该说也并非不可能，只是这样做伴随着一定的成本。从技术革新的角度来看，如果制造出昂贵的 CPU 也能卖得出去，这样的环境才是理想的。当然，总有一些领域，如 3D 图形、视频编码、物理计算等，即便再强大的 CPU 也不够用。但是这样的领域毕竟有限。每年不断高涨的技术革新成本到底该如何筹措，还是应该放弃技术革新从竞争中退出？近年来受到全球经济形势低迷的影响，半导体制造商们也面临着这一艰难的抉择。

超越极限

正如之前讲过的，摩尔定律已经接近极限，这是不争的事实。退一万步说，即使集成电路的精密化真的能够按现有的速度一直演进下去，总有一天一个晶体管会变得比一个原子还小。

不过，距离这一终极极限尚且还有一定的余地。现在我们所面临的课题，解决起来的确很有难度，但并没有到达无法克服的地步。

首先，关于导线宽度的问题，运用远紫外线和 X 射线的工艺已经处于研发阶段。由于这些波长极短的光源难以掌控，因此装置会变得更大，成本也会变得更高。但反过来说，我们已经知道这样的做法是行得通的，剩下的事情只要花钱就能够解决了。

比较难以解决的是渗漏电流及其所伴随的发热问题。随着半导体工艺技术的改善，对于如何降低渗漏电流，也提出了很多种方案。例如通过在硅晶体中形成二氧化硅绝缘膜来降低渗漏电流的 SOI（Silicon On Insulator）等技术。此外，采用硅以外的材料来制造集成电路的技术也正在研究之中，但距离实用化还比较遥远。

从现阶段来看，要从根本上解决渗漏电流的问题是很困难的，但是像通过切断空闲核心和电路的供电来抑制耗电量（也就抑制了发热），以及关闭空闲核心并提升剩余核心工作主频（Hyper Boost）等技术，目前都已经实用化了。

不再有免费的午餐

看了上面的介绍，想必大家已经对摩尔定律，以及随之不断增加的晶体管能够造就何等快速的 CPU 有了一个大致的了解。现代的 CPU 中，通过大量晶体管来实现高速化的技术随处可见。

然而与此同时，我们印象中的 CPU 执行模型，与实际 CPU 内部的处理也已经大相径庭。由条件分支导致的流水线气泡，以及为了克服内存延迟所使用的高速缓存等，从 8086 时代的印象来看都是难以想象的。

而且，什么也不用考虑，随着时间的推移 CPU 自然会变得越来越快，这样的趋势也快要接近极限了。长期以来，软件开发者一直受到硬件进步的恩惠，即便不进行任何优化，随着计算机的更新换代，同样的价格所能够买到的性能也越来越高。不过，现在即便换了新的计算机，有时也并不能带来直接的性能提升。要想提升性能，则必须要积极运用多核以及 CPU 的新特性。

最近，GPGPU（General Purpose GPU，即将 GPU 用于图形处理之外的通用编程）受到了越来越多的关注，由于 GPU 与传统 CPU 的计算模型有着本质的区别，因此需要采用专门的编程技术。

即便什么都不做，CPU 也会变得越来越快的时代结束了，今后为了活用新的硬件，软件开发者必须要付出更多的努力——这样的情况，我将其称为“免费午餐的终结”。

在未来的软件开发中，如果不能了解 CPU 的新趋势，就无法提高性能。新的计算设备必然需要新的计算模型，而这样的时代已经到来。

6.2 UNIX 管道[①]

诞生于 20 世纪 60 年代后半的 UNIX，与之前的操作系统相比，具有一些独到的特点，其中之一就是文件的结构。在 UNIX 之前，大多数操作系统中的文件指的是结构化文件。如果熟悉 COBOL 的话解释起来会容易一些，所谓结构化文件就是拥有结构的记录的罗列（图 1）。

但 UNIX 的设计方针是重视简洁，因此在 UNIX 中抛弃了对文件本身赋予结构的做法，而是将文件定义为单纯的字节流。对于这些字节流应当如何解释，则交给每个应用程序来负责。文件的内容是文本还是二进制，也并没有任何区别。

例如，图 1 中所示的平面文件（flat file），是采用每行一条记录、记录的成员之间用逗号进行分隔的 CSV（Comma Separated Values）格式来表现数据的。这并不是说 UNIX 对 CSV 这种文件格式有特别的规定，而只是相应的应用程序能够对平面文件中存放的 CSV 数据进行解释而已。

结构化文件

姓名	松本行弘
地址	松江市
电话号码	0852-28-××××
员工编号	7
基本工资	200000
姓名	笹田耕一
地址	东京都
电话号码	03-3855-××××
员工编号	33
基本工资	180000
:	:

平面文件（CSV）

```
姓名，地址，电话号码，员工编号，基本工资
松本行弘，岛根县，0852-28-XXXX,0007,200000
笹田耕一，东京都，03-3855-XXXX,0033,180000
…
```

图 1　结构化文件与平面文件

UNIX 的另一个独到之处就是 Shell。Shell 是 UNIX 用来和用户进行交互的界面，同时也是能够将命令批处理化的一种语言。

在 UNIX 之前的操作系统中，也有类似的命令管理语言，如 JCL（Job Control Language）。但和 JCL 相比，Shell 作为编程语言的功能更加丰富，可以对多个命令进行灵活地组合。如果要重复执行同样的操作，只要将操作过程记录到文件中，就能够很容易地作为程序来执行。像这样由“执行记录”生成的程序，被称为脚本（script），这也是之后脚本语言（script language）这一名称的辞源。

① 这里的“管道”和上一节中的“流水线”都来自同一个英文单词 pipeline，它们在本质上的含义也是相同的，只是在不同的领域中习惯译法不同。

在 UNIX 中至今依然存在 script 这个命令，这个命令的功能是将用户在 Shell 中的输入内容记录到文件中，根据所记录的内容可以编写出脚本程序。script 一词原本是“剧本”的意思，不过命令行输入是一种即兴的记录，也许叫做 improvisation（即兴表演）更加合适。

最后一点就是串流管道（stream pipeline）。UNIX 进程都具有标准输入和标准输出（还有标准错误输出）等默认的输入输出目标，而 Shell 在启动命令时可以对这些输入输出目标进行连接和替换。通过这样的方式，就可以将某个命令的输出作为另一个命令的输入，并将输出进一步作为另一个命令的输入，也就是实现了命令的“串联”。

在现代的我们看来，这三个特征都已经是司空见惯了的，但可以想象，在 UNIX 诞生之初，这些特征可是相当创新的。

管道编程

下面我们来看一看运用了串流管道的实际程序。图 2 是经常被用作 MapReduce[①]例题的用于统计文件中单词个数的程序。

通过这个程序来读取 Ruby 的 README 文件，会输出图 3 这样的结果。

```
tr -c '[:alnum:]' '¥n' | grep -v '^[0-9]*$' | sort | uniq -c | sort -r
```

图 2　单词计数程序

Shell 中，用“|”连接的命令，其标准输出和标准输入会被连接起来形成一个管道。这个程序是由以下 5 条命令组成的管道。

```
1. tr -c '[:alnum:]' '\n'
2. grep -v '^[0-9]*$'
3. sort
4. uniq -c
5. sort -r
```

下面我们来具体讲解一下每个命令的功能。

“tr”是 translate 的缩写，其功能是将输入的数据进行字符替换。tr 会将第一个参数所指定的字符集合（这里的 [:alnum:] 表示字母及数字的意思）用第二个参数所指定的字符进行替换。“-c（complement）”选项的意思是反转匹配，整体来看这条命令的功能就是“将除字母和数字以外的字符替换成换行符”。

① 一种通过分布式并行处理对庞大数据库进行高速访问的手法，分为“Map”和“Reduce”两个步骤进行处理。（原书注）

grep 命令用来搜索与模板相匹配的行。在这里，模板是通过正则表达式[①]来指定的：

```
^[0-9]*$
```

这里，“^”表示匹配行首，“$”表示匹配行尾，“[0-9]*”表示匹配“0 个或多个数字组成的字符串”。结果，这一模板所匹配的是“只有数字的行或者是空行”。

-v（revert）选项表示反转匹配，也就是显示不匹配的行。因此，这条 grep 命令的执行结果是“删除空行或者只有数字的行”。

之前的 tr 命令已经将字母和数字之外的字符全部替换成了换行符，也就是说将符号、空格等全部转换成了只有一个换行符的行（即空行）。对空行计数是没有意义的，因此需要忽略这些空行。此外，只有数字的行也不能算是单词，因此也需要忽略。

接下来的 sort 是对行进行重新排序的命令。到这条命令之前，数据流已经被转换成每行一个单词的排列形式，通过 sort 命令可以对原文中出现的单词按照字母顺序进行排序。这一排序操作看似没什么用，但接下来我们需要用 uniq 命令去掉重复的行，因此必须事先对输入的数据流进行排序。

```
     33 ruby
     23 the
     19 to
     16 prefix
     16 DESTDIR
     13 and
     13 Ruby
     11 lib
     11 is
     11 TEENY
     11 MINOR
     11 MAJOR
      7 of
      6 you
      6 org
      6 lang
      6 in
      6 be
      5 not
（中略）

      1 Author
      1 Aug
      1 Advanced
```

图 3　单词计数结果（节选）

uniq 是 unique 的缩写，该命令可以从已排序的文件中去掉重复的行。-c（count）选项表示在去掉重复行的同时显示重复的行数。在这里我们输入的文件是每行一个单词的形式，因此统计出已排序的单词序列中重复的行数，也就相当于是统计出了单词的数量。uniq 命令才是单词计数的本质部分。

最后我们用 sort -r 命令对输出的信息进行整形。uniq 命令执行完毕之后，就完成了“统计单词数量”这一任务，但从人类的角度来看，将单词按出现的数量降序排列才是最自然的，因此我们再执行一次 sort 命令。

我们希望在查看统计结果时将出现数量最多的单词（可以认为是比较重要的单词）放在前面，因此这次我们对 sort 命令加上了 -r（reverse）选项，这个选项代表降序排列的意思。这个命令有一个副作用，就是出现数量相同的单词，会被按照字母逆序排列，这一点就请大家多多包涵吧。

① 一种用特殊字符描述字符串匹配模板的手法。（原书注）

将单词按出现数量降序排列的同时，还要将出现数相同的单词按字母顺序排列，实现起来是出乎意料地麻烦。这里就当是给各位读者留个思考题吧。其实用 Ruby 和 Awk 就可以比较容易地解决这个问题了。

像上面这样,将完成一个个简单任务的命令组合起来形成管道,就可以完成各种各样的工作,这就是 UNIX 范儿的管道编程。

多核时代的管道

在 UNIX 诞生的 20 世纪 60 年代末，多核 CPU 还不存在，因此管道原本的设计也并非以运用多核为前提。然而，不知是偶然还是必然，管道对于多核的运用却是非常有效的。

下面我们来看看在多核环境中，管道的执行是何等高效。

首先，我们来思考一下非常原始的单任务操作系统，例如 MS-DOS。说是“原始”，但其实 MS-DOS 相比 UNIX 来说算是非常年轻的，在这里我们先忽略这一点吧。在 MS-DOS 中，同时只能有一个进程在工作，因此管道是通过临时文件来实现的。例如，当执行下列管道命令时：

```
command-a | command-b
```



图 4　单任务操作系统的管道

MS-DOS（准确地说应该是相当于 Shell 的 command.com）会生成一个临时文件，并将“command-a”的输出结果写入文件中。command-a 的执行结束之后，再以该临时文件作为输入源来执行“command-b”。由于 MS-DOS 是一个单任务操作系统，每次只能进行一项处理，当然也就无法对多核进行运用（图 4）。

接下来我们来思考一下单核环境下的多任务操作系统。在这样的环境下，管道的命令是并行执行的。但由于只有一个核心，因此无法做到完全同时进行。和刚才一样，执行下列命令：

```
command-a | command-b
```

这次 command-a 与 command-b 是同时启动的。

然后，进程会在不断相互切换中各自执行，command-b 会进入等待输入的状态。当 command-b 为读取数据发出系统调用时，如果暂时没有立即可供读取的数据，则操作系统会在数据准备好之前暂停 command-b 的进程，并使其休眠。

另一方面，command-a 继续执行，其结果会输出到某个地方。这样一来 command-b 就有了可供读取的数据，command-b 的进程就会被唤醒并恢复执行。

像这样，数据输出以接力棒的形式进行运作，多个进程交替工作，就是单核多任务环境中的执行方式（图 5）。

和单任务相比，多任务环境下的优势在于没有了无谓的文件输入输出操作，从而削减了相应的开销。

而且，由于多个进程是依次执行的，先得出的结果会立即通过管道传递，因此获取结果也会比较快一些。

不过，在多任务环境下，进程的切换也需要一定的开销，从总体来看，执行时间也未必会缩短。

接下来终于要讲到多核环境下的管道了。简单起见，在这里我们假设将 command-a 和 command-b 分别分配给两个不同的核心，在这样的情况下，管道执行如图 6 所示。

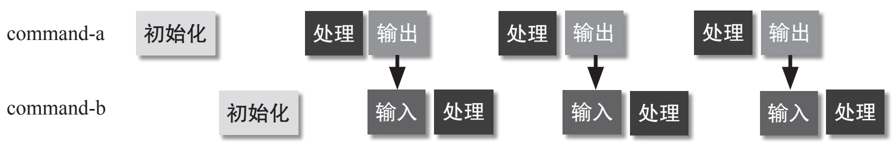


图 5　多任务操作系统的管道（单核）

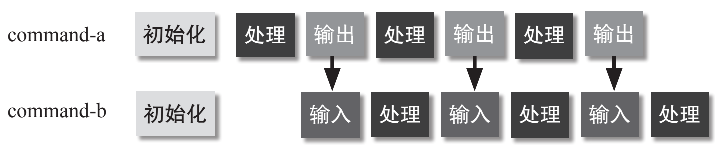


图 6　多任务操作系统的管道（多核）

我们可以看出，和图 5 相比，同时执行的部分增多了。非常粗略地数了一下，图 4 中需要 11 步完成的处理，这里只需要 8 步就完成了。不过我们投入了两个核心，理想状态下应该比单核缩短一半，但这样的理想状态是很难实现的。

假设操作系统足够聪明的前提下，只要增加管道的级数，使能够重叠的部分也相应增加，即便不特意去管多个核心的配置，只要自然编写程序形成管道，操作系统就会自动利用多个核心来提高处理能力。之所以说串流管道是非常适合多核的一种编程模型，原因也正是在于此。

xargs——另一种运用核心的方式

大家知道 xargs 这个命令吗？ xargs 是用于将标准输入转换成命令行参数的命令。

例如，要在当前目录下搜索所有文件名中以“~”结尾的文件，需要执行 find 命令：

```
# find . -name '*~' ⏎
```

⏎表示换行

这样就会将符合条件的文件名在标准输出中列出。

那么，如果我要将这些文件全部删除的话又该怎么做呢？这时就该轮到 xargs 命令出场了。

```
# find . -name '*~'|xargs rm ⏎
```

这样一来，传递到 xargs 标准输入的文件名列表就作为命令行参数传递给了 rm 命令，于是就删除了符合条件的所有文件。

还有一个很少有人会实际碰到的问题，那就是命令行参数的数量是有上限的，如果传递的参数过多，命令执行就会失败。xargs 也考虑了这一点，当参数过多时会分成几条命令分别执行。

上面所讲的内容与多核没什么关系，不过 xargs 提供了一个用于多核的命令行参数“-P”。

如图 7 所示，是用于将当前目录下未压缩的（即扩展名不是 .gz 的）文件全部进行压缩的管道命令。

```
# find . \! -name *.gz -type f -print0 | xargs -null -P 4 -r gzip -9 ⏎
```

图 7　文件压缩管道命令

首先是 find 命令，它的含义如下：

- “.”表示当前目录下
- “\! -name *.gz”表示文件名不以 .gz 结尾
- “-type f”表示一般文件（而不是目录等特殊文件）
- “-print0”表示将符合上述条件的文件名打印到标准输出。为了应对包含空格的文件名，采用 null 作为分隔符。

这样我们就得到了“当前目录下未压缩的文件名列表”。得到该列表之后，xargs 命令被执行。xargs 命令中的“-P”选项，表示同时启动指定数量的进程，这里我们设定为同时执行 4 个进程。“-r”选项表示当输入为空时不启动命令，即当不存在符合条件的文件时就表示不用进行压缩，因此我们在这里使用了“-r”选项。

为了应付空格，find 命令使用了“-print0”选项，相应地，必须同时使用“-null”选项。通过这样的操作，就实现了将要压缩的对象文件名作为参数传递给“gzip -9”命令来执行。

gzip 命令的“-9”选项表示使用较高的压缩率（会花费更多的时间）。

我们知道，文件的压缩比单纯的输入输出要更加耗时，而且，多个文件的压缩操作之间没有相互依赖的关系，这些操作是相互独立进行的。对于这样的操作，如果能够分配到多个进程来同时进行，应该说是最适合多核环境的工作方式。

在多核环境中，是否对 xargs 命令使用“-P”选项，直接影响了处理所需要的时间。由于 gzip 命令的输入输出等操作也需要一定的处理时间，因此 -P 设定的进程数应该略大于实际的核心数。我用手上的双核电脑进行了测试，用两个核心设定 4 个进程来执行时，可以获得最高的性能。

不过，在我所做的测试中，当文件数量较少时，即便使用了 -P 选项，也只能启动一个进程，无法充分利用多核。在这种情况下，对 xargs 命令使用 -n 选项来设定 gzip 一次性处理的文件数量，也许是个好主意。

例如，如果使用“-n 10”选项，就可以对每 10 个文件启动一个 gzip 进程。在我所做的测试中，启动 4 个进程进行并行压缩时，处理速度可以提高大约 40%。理想状态下，两个核心应该可以得到 100% 的性能提升，因此 40% 的成绩比我预想的要低。当然，这也说明在实际的处理中，有很大一部分输入输出的开销是无法通过增加核心数量来弥补的。

注意瓶颈

在这里需要注意的是，瓶颈到底发生在哪里。

多核环境是将任务分配给多个 CPU 来提高单位时间处理能力的一种手段。也就是说，只有当 CPU 能力成为处理瓶颈时，这一手段才能有效改善性能。

然而，一般的多核计算机上，尽管搭载了多个 CPU，但其他设备，如内存、磁盘、网络设备等是共享的。当处理的瓶颈存在于 CPU 之外的这些地方时，即便投入多个核心，也丝毫无法改善性能。

在这种情况下，我们需要的不仅是多个 CPU，而是由多台“计算机”组成的分布式计算环境。分布式计算也是一项相当重要的技术，我们在这里不再过多赘述。

阿姆达尔定律

阿姆达尔定律是一个估算通过多核并行能够获得多少性能提升的经验法则，是由吉恩 · 阿

姆达尔（Gene Amdahl，1922 ~ ）提出的，它的内容是：

> （通过并行计算所获得的）系统性能提升效果，会随着无法并行的部分而产生饱和。

正如在刚才 xargs 的示例中所遇到的，即便是多核计算机，一般也只有一个输入输出控制器，而这个部分无法获得并行计算所带来的效果，很容易成为瓶颈。

而且，当数据之间存在相互依赖关系时，在所依赖的数据准备好之前，即便有空闲的核心也无法开始工作，这也会成为瓶颈。

综上所述，大多数的处理都不具备“只要增加核心就能够提高速度”这一良好的性质，这一点与在 CPU 内部实现流水线的艰辛似乎存在一定的相似性。

根据阿姆达尔定律，并行化之后的速度提升比例可以通过图 8 的公式来估算。假设 N 为无穷大，速度的提升最多也只能达到：

```
1 /(1 - P)
```

例如，即便在 P 为 90% 这一非常理想的情况下，无论如何提高并行程度，整体上最多能够获得的性能提升也无法超过基准的 10 倍。这是因为，“(1 - P)”所代表的无法并行化的部分成为了瓶颈，使得并行化效果存在极限。

$$\text{速度提升比例} = \frac{1}{(1-P) + P/N}$$

P = 可并行化部分的占比（相对于基准运算时间而言可并行处理部分的比例）
N = 并行度（处理器数量）

图 8　并行化后速度提升比例的公式

多核编译

像我们这些工程师在用电脑时，最消耗 CPU 的工作恐怕就是编译了。当然，编译也伴随一定的输入输出操作，但预处理、语法解析、优化、代码生成等操作对于 CPU 的开销是相当大的。

要编译一个文件，首先需要将 C 语言源文件（*.c）进行预处理（cpp）。cpp 会进行头文件（*.h）加载（#include）、宏定义（#define）、宏展开等操作。

cpp 的运行结果被送至编译器主体（cc1）。cc1 会进行语句、语法解析和代码优化，并输出汇编文件（*.s）。随后，汇编器会将汇编文件转换为对象文件（*.o），也有些编译器可以不通过汇编器直接输出对象文件。

当每个 C 语言源文件都完成编译，并生成相应的对象文件之后，就可以启动连接器（ld）来生成最终的可执行文件了。连接器会将对象文件与各种库文件（静态链接库 *.a 和动态链接库 *.so）进行连接（某些情况下还会进行一些优化），并输出最终的可执行文件（图 9）。

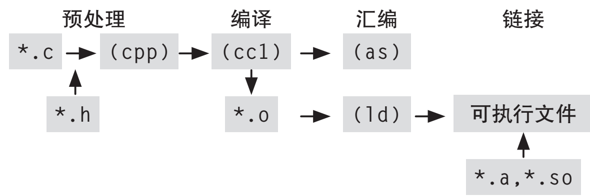


图 9　C 语言编译流程

UNIX 的“make”工具中提供了一个正好可以用于多核的选项——“-j（jobs）”，通过这个选项可以设定同时执行的进程数量。例如：

```
# make -j4 ⏎
```

就表示用 4 个线程进行并行编译。从过去的经验来看，-j 的设置应该略大于实际的核心数量为佳。

ccache

我们先放下多核的话题，说点别的。有一个叫做 ccache 的工具，可以有效提高编译的速度。ccache 是通过将编译结果进行缓存，来减少再次编译的工作量，从而提高编译速度的。

使用方法很简单，编译时在编译器名称前面加上 ccache 即可。例如：

```
# CC='ccache gcc' make -j4 ⏎
```

这样就可以让再次编译时所需的时间大幅缩短。每次都指定的话比较麻烦，也可以一开始就写到 Makefile 中。

当源代码或者其所依赖的头文件等发生修改时，make 会重新执行编译。不过，源文件中也有很多行是和实际变更的部分无关的，而 ccache 会将（以函数为单位的）编译结果保存在主目录下的“.ccache”目录中，然后，在实际执行编译之前，与过去的编译结果进行比较，如果源代码的相应部分没有发生修改，则直接使用过去的编译结果。

在 CPU 中，缓存是高速化的一个重要手段，而在改善编译速度的场景中，也可以应用缓存技术。像这样，类似的手段出现在各种不同的场景中，的确是很有意思的事情。

distcc

还有其他一些改善编译速度的方法，例如 distcc 就是一种利用多台计算机来改善编译速度的工具。

和 ccache 一样，只要在编译器名称前面加上 distcc 就可以改善编译性能了。不过在此之前，需要先配置好用哪些计算机来执行编译操作。在下列配置文件中：

```
# ~/.distcc/hosts ⏎
```

填写用于执行编译的主机名（多个主机名之间用逗号分隔）。

当然，并不是随便填哪台主机都可以的。基本上，用于执行编译的主机应该是启动了 distccd 服务的主机，或者是可以通过 ssh 来登录的主机才行。启动 distccd 服务的主机直接填写主机名，可在 ssh 登录的主机前面加上一个“@”。当登录的用户名和本机不同时，需要在 @ 前面写上用户名。

通过 ssh 来执行会提高安全性（distccd 没有认证机制），但由于加密等带来的开销，编译性能会下降 25% 左右，因此用户需要在性能、安全性和易用性之间做出选择。

准备妥当之后，执行：

```
# CC='distcc gcc' make -j4 ⏎
```

就可以实现分布式编译了。

distcc 的伟大之处在于，虽然是分布式编译，但无需拥有所有的头文件和库文件等完整的环境，只要（在同一个 CPU 下）安装了编译器，并能够运行 ssh 的主机，就可以很容易地实现分布式编译。之所以能够实现这一点，秘密在于预处理器和连接器是在本地执行的，而发送给远程主机的是已经完成预处理的文件。

编译性能测试

那么，通过使用上述这些手段，到底能够对编译性能带来多大的改善呢？我们来实际测试一下。

表 1 显示了运用各种手段后的测试结果，其中编译的对象是最新版的 Ruby 解释器。用于执行编译的是我那台有些古老的爱机——ThinkPad X61 Core2 duo 2.2GHz（双核）。distcc 分布式编译使用的是一台 Quad-Core AMD Opteron 2.4GHz（四核）的计算机。

表1　编译性能测试

编译条件	编译时间（秒）
仅 gcc -j1	18.464
仅 gcc -j2	10.611
仅 gcc -j4	10.823
仅 gcc -j8	11.006
ccache -j1	20.874
ccache -j1（第2次）	0.454
distcc -j2	11.649
distcc -j4	7.138
distcc -j8	7.548

使用未经过任何优化的 gcc 进行编译时，整个编译过程需要约 18.5 秒。使用 make 的 -j 选项启动多个进程时，

由于充分利用了两个核心，使得速度提高了 40% 以上。

ccache 首次执行时比通常情况还要慢一点，但由于编译结果被缓存起来，在删除对象文件之后，用完全相同的条件再次编译时，由于完全不需要执行实际的编译操作，只需要取出缓存的内容就可以完成处理，因此编译速度快得惊人。

distcc 的测试中只用了一台主机，在 make -j2 的情况下，由于 ssh 的开销较大，因此和本地执行相比性能改善不大，但如果设置更多的进程数量，执行时间就可以大大缩短。

小结

阿姆达尔定律指出，并行性是存在极限的，因此只靠多核无法解决所有的问题。但是大家应该能够看出，只要配合适当的编程技巧，还是能够比较容易地获得很好的效果。可以说，多核在将来还是颇有前途的。

6.3 非阻塞 I/O

在需要处理大量连接的服务器上，如果使用线程的话，内存负荷和线程切换的开销都会变得非常巨大。因此，监听“有输入进来”等事件并进行应对处理，采用单线程来实现会更加高效。像这样通过“事件及应对处理”的方式来工作的软件架构，被称为事件驱动模型（event driven model）。

这种模型虽然可以提高效率，但也有缺点。在采用单线程来进行处理的情况下，当事件处理过程中由于某些原因需要进行等待时，程序整体就会停止运行。这也就意味着即便产生了新的事件，也无法进行应对了。

像这样处理发生停滞的情况被称为阻塞。阻塞多半会在等待输入输出的时候发生。对于事件驱动型程序来说，阻塞是应当极力避免的。

何为非阻塞 I/O

由于大部分输入输出操作都免不了会遇到阻塞，因此在输入输出时需要尤其注意。输入输出操作速度并不快，因此需要进行缓冲。当数据到达缓冲区时，读取操作只需要从缓冲区中将数据复制出来就可以了。

在缓冲机制中，有两种情况会产生等待。一种是当缓冲区为空时，需要等待数据到达缓冲区（读取时）；另一种是在缓冲区已满时，需要等待缓冲区腾出空间（写入时）（图 1）。这两种“等待”就相当于程序停止工作的“阻塞”状态。

尤其是在输入（读取）时，如果在数据到达前试图执行读取操作，就会一直等待数据的到达，这样肯定会发生阻塞。

相比之下，输出时由于磁盘写入、网络传输等因素，也有可能会发生阻塞，但发生的概率并不高。而且即便发生了阻塞，等待时间也相对较短，因此不必过于在意。

要实现非阻塞的读取操作，有下列几种方法。

- ❑ 使用 read(2) 的方法
- ❑ 使用 read(2)+select 的方法
- ❑ 使用 read(2)+O_NONBLOCK 标志的方法
- ❑ 使用 aio_read 的方法
- ❑ 使用信号驱动 I/O 的方法

这些方法各有各的优缺点，我们来逐一讲解一下。

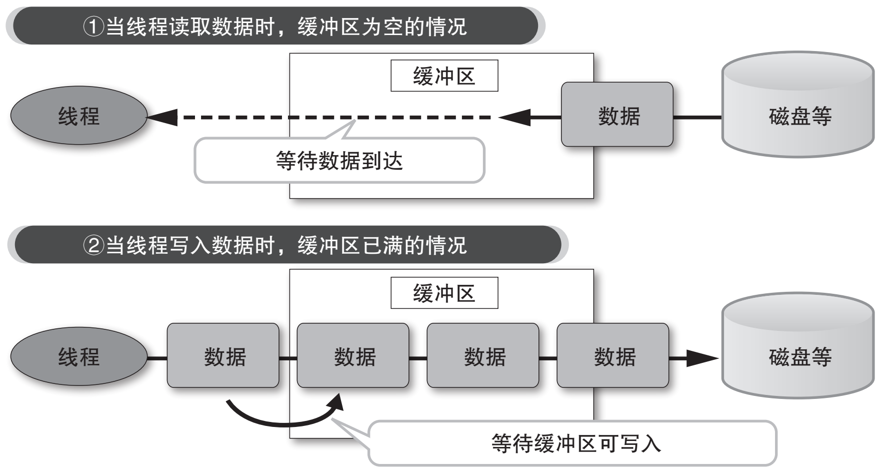


图 1　输入输出中发生阻塞的原因

使用 read(2) 的方法

首先，我们先来确定示例程序的结构。在这里，我们只写出了程序中实际负责读取处理的回调部分。

我们将回调函数命名为 callback，它的参数用于读取的文件描述符（int fd）和注册回调函数时指定的指针（void *data）（图 2）。关于输出，我们再设置一个 output 函数。

在实际的程序中，需要在事件循环内使用选择可读写文件描述符的“select 系统调用”和“epoll”等，对文件描述符进行监视，并对数据到达的文件描述符调用相应的回调函数。

```
int
callback(int fd, void *data)
{
  ....
  /* 返回值成功为1 */
  /* 到达EOF为0 */
  /* 失败为-1 */
}

void
output(int fd, const char *p, int
len)
{
  ....
}
```

图 2　回调函数与输出函数

我们先来看看只使用 read 系统调用的实现方法。对了，所谓 read(2)，是 UNIX 中广泛使用的一种记法，代表“手册显示命令 man 的第 2 节中的 read”的意思。由于第 2 节是系统调用，因此可以认为 read(2) 相当于“read 系统调用”的缩写。

只使用 read(2) 构建的回调函数如图 3 所示。

```
void
callback(int fd, void *data)
{
  char buf[BUFSIZ];
  int n;

  n = read(fd, buf, BUFSIZ);
  if (n < 0) return -1;  /* 失败 */
  if (n == 0) return 0;  /* EOF */
  output(fd, buf, n);    /* 写入 */
  return 1;              /* 成功 */
}
```

图 3　用 read(2) 实现的输入操作（ver.1）

程序非常简单。当这个回调函数被调用时，显然输入数据已经到达了，因此只要调用 read 系统调用，将积累在输入缓冲区中的数据复制到 buf 中即可。当输入数据到达时，read 系统调用不会发生阻塞。

read 系统调用的功能是：①失败时返回负数；②到达 EOF 时返回 0；③读取成功时返回读取的数据长度。只要明白了这些，就很容易理解图 2 中程序的行为了吧。小菜一碟。

不过，这样简单的实现版本中必然隐藏着问题，你发现了吗？这个回调函数正确工作的前提是，输入数据的长度要小于 BUFSIZ（C 语言标准 IO 库中定义的一个常量，值貌似是 8192）。

```
void
callback(int fd, void *data)
{
  char buf[BUFSIZ];
  int n;

  for (;;) {  /*
    n = read(fd, buf, BUFSIZ);
    if (n < 0) return -1;  /* 失败 */
    if (n == 0) return 0;  /* EOF */
    output(fd, buf, n);          /* 写入
*/
    if (n < BUFSIZ) break; /* 读取完毕,退出
*/
  }
  return 1;              /* 成功 */
}
```

图 4　用 read(2) 实现的输入操作（ver.2）

但是，在通信中所使用的数据长度一般都不是固定的，某些情况下需要读取的数据长度可能会超过 BUFSIZ。于是，能够支持读取长度超过 BUFSIZ 数据的版本如图 4 所示。

在版本 2 中，当读取到的数据长度小于 BUFSIZ 时，也就是当输入缓冲区中的数据已经全部读取出来的时候，程序结束。当读取到的数据长度等于 BUFSIZ 时，则表示缓冲区中还可能有残留的数据，因而可通过反复循环，直到读取完毕为止。

问题都解决了吗？还没有，事情可没那么简单。当输入的数据长度正好等于 BUFSIZ 时，这个程序会发生阻塞。我们说过，避免阻塞对于回调函数来说是非常重要的，因此这个程序还无法实际使用，我们还需要进行一些改进。

边沿触发与电平触发

好了，接下来我要宣布一件重要的事。我们刚才说图 3 的程序只能支持读取长度小于 BUFSIZ 的数据，但其实只要将读取的数据直接输出，它还是可以正常工作的，而且不会发生阻塞。不过，要实现这一点，负责事件监视的部分需要满足一定的条件。

在事件监视中，对事件的检测方法有两种，即边沿触发（edge trigger）和电平触发（level trigger）。这两个词原本是用在机械控制领域中的，边沿触发是指只在状态变化的瞬间发出通知，而电平触发是指在状态发生变化的整个过程中都持续发出通知（图 5）。

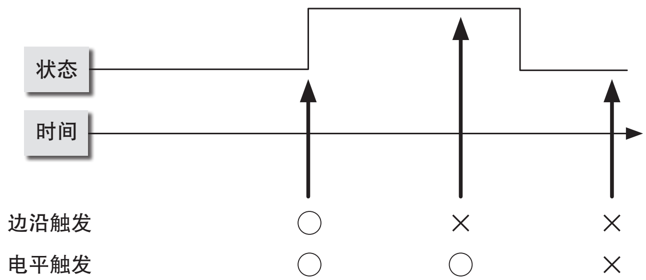


图 5　边沿触发与电平触发

select 系统调用属于电平触发，epoll 默认也是电平触发，但 epoll 也可以通过显式设置来实现边沿触发。

具体来说，是在 epoll_event 结构体的 events 字段通过 EPOLLET 标志来进行设置的。

要让图 3 的程序在不阻塞的状态下工作，事件监视就必须采用电平触发的方式。也就是说，在调用回调函数执行输入操作之后，如果读取缓冲区中还有残留的数据，在电平触发的方式下，就会再次调用回调函数来进行读取操作。

那么，采用电平触发就足够了吗？边沿触发的存在还有什么意义呢？由于边沿触发只在数据到达的瞬间产生事件，因此总体来看事件发生的次数会比较少，这也就意味着回调函数的启动次数也会比较少，可以提高效率。

使用 read(2) + select 的方法

刚才已经讲过，图 3 版本的程序，会将输入缓冲区中积累的数据全部读取出来，而当输入缓冲区为空时，调用 read 系统调用就会发生阻塞。为了避免这个问题，需要在调用 read 之前检查输入缓冲区是否为空。

下面，我们来创建一个 checking_read 函数。它先调用 read 系统调用，然后通过 select 系统调用检查输入缓冲区中是否有数据（图 6）。为了判断是否有剩余数据，checking_read 比 read 增

加了一个参数。调用 checking_read 来代替 read，如果参数 cont 的值为真，就表示输入缓冲区中还有剩余的数据。

用这种方法，在边沿触发的方式下也可以正常工作。边沿触发的好处就是能够减少事件发生的次数，但相对地，select 系统调用的调用次数却增加了。此外，在每次调用 read 系统调用时，还要问一下“还有剩下的数据吗”，总让人感觉怪怪的。

```
#include <sys/time.h>
#include <sys/types.h>

int
checking_read(int fd, char *buf, int len, int *cont)
{
  int n;

  *cont = 0;                /* 初始化 */
  n = read(fd, buf, len);   /* 调用read(2) */
  if (n > 0) {              /* 读取成功 */
    fd_set fds;
    struct timeval tv;
    int c;

    FD_ZERO(&fds);         /* 准备调用select(2) */
    FD_SET(fd, &fds);
    tv.tv_sec = 0;         /* 不会阻塞 */
    tv.tv_usec = 0;
    c = select(fd+1, &fds, NULL, NULL, &tv);
    if (c == 1) {        /* 返回值为1=缓冲区不为空 */
      *cont = 1;           /* 设置继续标志 */
    }
  }
  return n;
}

void
callback(int fd, void *data)
{
  char buf[BUFSIZ];
  int n, cont;

  for (;;) {  /*
    n = checking_read(fd, buf, BUFSIZ, &cont);
    if (n < 0) return -1;  /* 失败 */
    if (n == 0) return 0;  /* EOF */
    output(fd, buf, n);    /* 写入 */
    if (!cont) continue;   /* 读取完毕,退出 */
  }
  return 1;                 /* 成功 */
}
```

图 6　用 read（2）实现的输入操作（ver.3）

使用 read+O_NONBLOCK 标志

毕竟 read 系统调用可以直接接触输入缓冲区，那么理所当然地，在读取数据之后它应该知道缓冲区中是否还有剩余的内容。那么，能不能实现“调用 read，当会发生阻塞时通知我一下”这样的功能呢？

当然可以。只要在文件描述符中设置一个 O_NONBLOCK 标志，当输入输出中要发生阻塞时，系统调用就会产生一个“继续执行的话会发生阻塞”的错误消息提示，这个功能在 UNIX 系操作系统中是具备的。使用 O_NONBLOCK 的版本如图 7 所示。

```
#include <fcntl.h>
#include <errno.h>

/* (a) 初始化程序的某个地方 */
inf fl;

fl = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, fl|O_NONBLOCK);
/* 到此为止 */
```

```
void
callback(int fd, void *data)
{
  char buf[BUFSIZ];
  int n;

  for (;;) {  /*
    n = read(fd, buf, BUFSIZ);
    if (n < 0) {
      if (errno == EAGAIN) { /* EAGAIN=缓冲区为空 */
      return 1;               /* 读取操作结束 */
      }
      return -1;            /* 失败 */
    }
    if (n == 0) return 0;    /* EOF */
    output(fd, buf, n);            /* 写入 */
  }
}
```

图 7　用 read(2) 实现的输入操作（ver.4）

怎么样？由于这次我们能够从本来拥有信息的 read 直接收到通知，整体上看比图 6 的版本要简洁了许多。

这个功能仅在对文件描述符设置了 O_NONBLOCK 标志时才会有效，因此在对文件描述符进行初始化操作时，不要忘记使用图 7(a) 中的代码对标志进行设置。

这种方法效率高、代码简洁，可以说非常优秀，但有一点需要注意，那就是大多数输入输出程序在编写时都没有考虑到文件描述符设置了 O_NONBLOCK 标志的情况。

当设置了 O_NONBLOCK 标志的文件描述符有可能发生阻塞时，会返回一个错误，而不会发生实质上的阻塞。一般的输入输出程序都没有预想到这种行为，因此发生这样的错误就会被认为是读取失败，从而引发输入输出操作的整体失败。

使用 O_NONBLOCK 标志时，一定要注意文件描述符的使用。O_NONBLOCK 标志会继承给子进程，因此在使用 fork 的时候要尤其注意。以前曾经遇到过这样的 bug：①对标准输入设置了 O_NONBLOCK；②用 system 启动命令；③命令不支持 O_NONBLOCK，导致诡异的错误。那时，由于忘记了子进程会继承 O_NONBLOCK 标志这件事，结果花了大量的时间才找到错误的原因。

Ruby 的非阻塞 I/O

刚才我们对 C 语言中的非阻塞 I/O 进行了介绍，下面我们来简单介绍一下 Ruby 的非阻塞 I/O。

Ruby 从 1.8.7 版本开始提供这里介绍过的两个实现非阻塞 I/O 的方法，即 read_partial 和 read_nonblock。

read_partial 方法可以将当前输入缓冲区中的数据全部读取出来。read_partial 可以指定读取数据的最大长度，其使用方法是：

```
str = io.read_partial(1024)
```

read_partial 基本上不会发生阻塞，但若输入缓冲区为空且没有读取到 EOF 时会发生阻塞。也就是说，仅在一开始原本就没有数据到达的情况下会发生阻塞。

换句话说，只要是通过事件所触发的回调中，使用 read_partial 是肯定不会发生阻塞的，因此 read_partial 在实现上相当于 read+select 的组合。

将图 6 的程序改用 Ruby 进行的实现如图 8 所示。不过，和图 6 的程序不同的是，当数据大于指定的最大长度时不会循环读取。在读取的数据长度等于最大长度时，如果循环调用 read_partial 就有可能会发生阻塞。这真是个难题。

```
def callback(io, data)
  input = io.read_partial(4096)
  output(io, input)
end
```

图 8　使用 read_partial 的示例

相对地，read_nonblock 则相当于 read+O_NONBLOCK 的组合。read_nonblock 会对 io 设

置 O_NONBLOCK 并调用 read 系统调用。read_nonblock 在可能要发生阻塞时，会返回 IO：：WaitReadable 模块所包含的异常。

read_nonblock 的参数和 read_partial 是相同的。我们将图 7 的程序用 read_nonblock 改写成 Ruby 程序（图 9）。和 C 语言的版本相比，Ruby 版显得更简洁，而且 read_nonblock 会自动设置 O_NONBLOCK 标志，因此不需要进行特别的初始化操作。

```
def callback(io, data)
  loop do
    begin
      input = io.read_nonblock(4096)
      output(io, input)
    rescue IO::WaitReadable
      # 缓冲区为空了,结束
      return
    end
  end
end
```

图 9　使用 read_nonblock 的例子

使用 aio_read 的方法

POSIX[①]提供了用于异步 I/O 的“aio_XXXX”函数集（表 1）。例如，aio_read 用于以异步方式实现和 read 系统调用相同的功能。这里的 aio 就是异步 I/O（Asynchronous I/O）的缩写。

aio 函数集的功能，是将通常情况下会发生阻塞的系统调用（read、write、fsync）在后台进行执行。这些函数只负责发出执行系统调用的请求，因此肯定不会发生阻塞。

表1　异步I/O函数

名　称	功　能
aio_read	异步 read
aio_write	异步 write
aio_fsync	异步 fsync
aio_error	获取错误状态
aio_return	获取返回值
aio_cancel	请求取消
aio_suspend	请求等待

运用 aio_read 的最简单的示例程序如图 10 所示，它的功能非常简单：

- ❑ 打开文件
- ❑ 用 aio_read 发出读取请求
- ❑ 用 aio_suspend 等待执行结束
- ❑ 或者用 aio_error 检查执行结束
- ❑ 用 aio_return 获取返回值

下面我们来看看程序的具体内容。

① POSIX（Portable Operating System Interface X，可移植操作系统接口）是由电气电子工程师学会（IEEE）制定的一套在各种 UNIX 操作系统上 API 的相互关联标准，正式名称为 IEEE 1003。

```
1       /* 异步I/O所需头文件 */
2       #include <aio.h>
3
4       /* 其他头文件 */
5       #include <unistd.h>
6       #include <string.h>
7       #include <stdio.h>
8       #include <errno.h>
9
10      int
11      main()
12      {
13          struct aiocb cb;
14          const struct aiocb *cblist[1];
15          char buf[BUFSIZ];
16          int fd, n;
17
18          /* 准备文件描述符 */
19          fd = open("/tmp/a.c", O_RDONLY);
20
21          /* 初始化控制块结构体 */
22          memset(&cb, 0, sizeof(struct aiocb)); /* 清空 */
23          cb.aio_fildes = fd;                        /* 设置fd */
24          cb.aio_buf = buf;                 /* 设置buf */
25          cb.aio_nbytes = BUFSIZ;           /* 设置buf长度 */
26
27          n = aio_read(&cb);               /* 请求 */
28          if (n < 0) perror("aio_read");     /* 请求失败检查 */
29
30      #if 1
31          /* 使用aio_suspend检查请求完成 */
32          cblist[0] = &cb;
33          n = aio_suspend(cblist, 1, NULL);
34      #else 1
35          /* 使用aio_error也能检查执行完成情况 */
36          /* 未完成时返回EINPROGRESS */
37          while (aio_error(&cb) == EINPROGRESS)
38              printf("retry\n");
39      #endif
40
41          /* 执行完成,取出系统调用的返回值 */
42          n = aio_return(&cb);
43          if (n < 0) perror("aio_return");
44
45          /* 读取的数据保存在aio_buf中 */
46          printf("%d %d ---\n%.*s", n, cb.aio_nbytes, cb.aio_nbytes, cb.aio_buf);
47          return 0;
48      }
```

图 10　异步 I/O 示例

第 19 行将文件 open 并准备文件描述符。不过，这只是一个例子，并没有什么意义，因为实际的异步 I/O 往往是以套接字为对象的。根据我查到的资料来看，像 HP-UX 等系统中，aio_

read 甚至是只支持套接字的。

从第 22 行开始对作为 aio_read 等的参数使用的控制块（aiocb）结构体进行初始化操作。read 系统调用有 3 个参数：文件描述符、读取缓冲区、缓冲区长度，但 aio_read 则是将上述这些参数分别通过 aiocb 结构体的 aio_fildes、aio_buf、aio_nbytes 这 3 个成员来进行设置。aiocb 结构体中还有其他一些成员，保险起见我们用 memset 将它们都初始化为 0（第 22 行）。

随后，我们使用 aiocb 结构体，通过 aio_read 函数预约执行 read 系统调用（第 27 行）。aio_read 只是提交一个请求，而不会等待读取过程的结束。对实际读取到的数据所做的处理，是在读取结束之后才进行的。

在这里我们使用 aio_suspend 执行挂起，直到所提交的任意一个请求执行完毕位置(第 33 行)。不过话说回来，我们也就提交了一个请求而已。

对请求执行完毕的检查也可以使用 aio_error 来实现。使用提交请求的 aiocb 结构体来调用 aio_error 函数，如果请求未完成则返回 EINPROGRESS，成功完成则返回 0，发生其他错误则返回相应的 errno 值。在这里，有一段对预处理器标明不执行的代码（34 ~ 39 行），这段代码使用 aio_error 用循环来检查请求是否完成。这是一个忙循环(busy loop)，会造成无谓的 CPU 消耗，因此在实际的代码中是不应该使用的。

读取请求完成之后，就该对读取到的数据进行处理了（42 ~ 46 行）。read 系统调用的返回值可以通过 aio_return 函数来获取。此外，读取到的数据会被保存到 aiocb 结构体的 aio_buf 成员所指向的数组中。

图 10 的程序中是使用 aio_suspend 和 aio_error 来检查请求是否完成的，其实异步 I/O 也提供了在读取完成时调用回调函数的功能。在回调函数的调用上，有信号和线程两种方式，下面(为了简单起见）我们来介绍使用线程进行回调的方式（图 11）。

图 11 的程序和图 10 的程序基本上是相同的，不同点在于，回调函数的指定（48 ~ 50 行）、回调函数（10 ~ 31 行）以及叫处理转交给回调函数并停止线程的 select（第 56 行）。

```
1       /* 异步I/O所需头文件 */
2       #include <aio.h>
3
4       /* 其他头文件 */
5       #include <unistd.h>
6       #include <string.h>
7       #include <stdio.h>
8       #include <errno.h>
```

图 11　异步 I/O 示例（回调）

```

static void
read_done(sigval_t sigval)
{
  struct aiocb *cb;
  int n;

  cb = (struct aiocb*)sigval.sival_ptr;

  /* 检查请求的错误状态 */
  if (aio_error(cb) == 0) {
    /* 获取已完成的系统调用的返回值 */
    n = aio_return(cb);
    if (n < 0) perror("aio_return");

    /* 读取到的数据存放在aio_buf中 */
    printf("%d %d ---\n%.*s", n, cb->aio_nbytes, cb->aio_nbytes, cb->aio_buf);

    /* 示例到此结束 */
    exit(0);
  }
  return;
}

int
main()
{
    struct aiocb cb;
    char buf[BUFSIZ];
    int fd, n;

    /* 准备文件描述符 */
    fd = open("/tmp/a.c", O_RDONLY);

    /* 初始化控制块结构体 */
    memset(&cb, 0, sizeof(struct aiocb)); /* 清空 */
    cb.aio_fildes = fd;                         /* 设置fd */
    cb.aio_buf = buf;                     /* 设置buf */
    cb.aio_nbytes = BUFSIZ;               /* 设置buf长度 */
    cb.aio_sigevent.sigev_notify = SIGEV_THREAD;
    cb.aio_sigevent.sigev_notify_function = read_done;
    cb.aio_sigevent.sigev_value.sival_ptr = &cb;

    n = aio_read(&cb);                    /* 请求 */
    if (n < 0) perror("aio_read");        /* 请求失败检查 */

    /* 停止线程,处理转交给回调函数 */
    select(0, NULL, NULL, NULL, NULL);
    return 0;
}
```

图 11　异步 I/O 示例（续）

```
/* 异步I/O所需头文件 */
#include <aio.h>

/* 其他头文件 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>

static void
read_done(sigval_t sigval)
{
  struct aiocb *cb;
  int n;

  cb = (struct aiocb*)sigval.sival_ptr;

  if (aio_error(cb) == 0) {
    /* 获取完成的系统调用的返回值 */
    n = aio_return(cb);
    if (n == 0) {               /* EOF */
      printf("client %d gone\n", cb->aio_fildes);
      aio_cancel(cb->aio_fildes, cb); /* 取消提交的请求 */
      close(cb->aio_fildes);  /* 关闭fd */
      free(cb);                        /* 释放cb结构体 */
      return;
    }
    printf("client %d (%d)\n", cb->aio_fildes, n);
    /* 直接写回 */
    /* 读取到的数据存放在aio_buf中 */
    /* 严格来说write也可能阻塞,但这里我们先忽略这一点 */
    write(cb->aio_fildes, (void*)cb->aio_buf, n);
    aio_read(cb);
  }
  else {                      /* 错误 */
    perror("aio_return");
    return;
  }
  return;
}

static void
register_read(int fd)
{
  struct aiocb *cb;
  char *buf;

  printf("client %d registered\n", fd);
  cb = malloc(sizeof(struct aiocb));
  buf = malloc(BUFSIZ);
```

图 12　使用 aio_read 的 echo 服务器（节选）

```
  /* 初始化控制块结构体 */
  memset(cb, 0, sizeof(struct aiocb)); /* 清空 */
  cb->aio_fildes = fd;          /* 设置fd */
  cb->aio_buf = buf;                   /* 设置buf */
  cb->aio_nbytes = BUFSIZ;             /* 设置buf长度 */
  cb->aio_sigevent.sigev_notify = SIGEV_THREAD;
  cb->aio_sigevent.sigev_notify_function = read_done;
  cb->aio_sigevent.sigev_value.sival_ptr = cb;

  /* 提交请求 */
  aio_read(cb);
}

int
main()
{
  struct sockaddr_in addr;
  int s = socket(PF_INET, SOCK_STREAM, 0);

  addr.sin_family = AF_INET;
  addr.sin_addr.s_addr = INADDR_ANY;
  addr.sin_port = htons(9989);
  if (bind(s, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
    perror("bind");
    exit(1);
  }
  listen(s, 5);

  for (;;) {
    /* 接受来自客户端套接字的连接 */
    int c = accept(s, NULL, 0);
    if (c < 0) continue;

    /* 开始监听客户端套接字 */
    register_read(c);
  }
}
```

图 12　使用 aio_read 的 echo 服务器（续）

由于这里我们只需要对回调函数的调用进行一次等待，因此在初始化完毕后用 select 进行等待，在回调函数中则使用 exit。实际的事件驱动服务器程序中，主程序应该是“接受客户端连接并注册回调函数”这样一个循环。此外，在回调函数的最后也不应该用 exit，而是通过 aio_read 再次提交 read 请求，用于再次读取数据。

如果用 SIGEV_THREAD 设置回调函数并调用 aio_read，从系统内部来看，实际上是用多个线程来实现异步 I/O。也就是说，回调函数是在独立于主线程的另一个单独的线程中执行的。因此，一旦使用了 SIGEV_THREAD，在回调函数中就不能调用非线程安全的函数。

只要是会修改全局状态的函数，都是非线程安全的。POSIX 标准库函数中也有很多是非线程安全的。如果在回调函数中直接或间接地调用了这些函数，就有可能引发意想不到的问题。

SIGEV_THREAD 是用线程来实现回调的，但并不是所有的输入处理都会使用独立的线程，因此不必担心线程数量会出现爆发性地增长。

最后，为了让大家对如何用异步 I/O 来编写事件驱动型程序有一个直观的印象，在图 12 中展示了一个使用了 aio_read 等手段实现的 echo 服务器的代码节选。通过上面的讲解，大家是否对不使用 libev 等手段的情况下，如何实现事件驱动编程有了一些了解呢？

Linux 中也提供了 io_getevents 等其他的异步 I/O 函数，这些函数的性能应该说更好一些。不过，它们都是 Linux 专用的，而且其事件响应只能支持通常文件，使用起来制约比较大，因此本书中的介绍，还是以 POSIX 中定义的，在各种平台上都能够使用的 aio_read 为主。

6.4 node.js

1990年，我大学毕业后进入软件开发公司工作，到现在已经有20年了，不由得感叹日月如梭。在这20年间，我从事软件开发工作的感受，就是无论是软件开发还是其他工作，最重要的就是提高效率。工作就像一座大山一样压在你面前，不解决掉它你就没饭吃。然而，自己所拥有的能力和时间都是有限的，要想在规定期限内解决所有的问题非常困难。

话说回来，在这20年的工作生涯中，我几乎没有开发过供客户直接使用的软件，这作为程序员似乎挺奇葩的。不过，我依然是一名程序员。从软件开发中，程序员能够学到很多丰富人生的东西。我从软件开发中学会了如何提高效率，作为应用，总结出了下面几个方法：

- 减负
- 拖延
- 委派

看起来这好像都是些浑水摸鱼的歪门邪道，其实这些方法对于提高工作效率是非常有用的。

减负

在计算机的历史上，提高处理速度的最有效手段，就是换一台新的电脑。计算机的性能不断提升，而且价格还越来越便宜，仅靠更新硬件就能够获得成倍的性能提升，这并不稀奇。

不过很遗憾，摩尔定律并不适用于人类，人类的能力不可能每两年就翻一倍，从工作的角度来看，上面的办法是行不通的。然而，如果你原地踏步的话，早晚会被更年轻、工资更便宜的程序员取代，效率先不说，至少项目的成本降低了，不过对于你来说这可不是什么值得高兴的事。

说点正经的，在软件开发中，如果不更换硬件，还可以用以下方法来改善软件的运行速度：

- 采用更好的算法
- 减少无谓的开销
- 用空间来换时间

如果将这些方法拿到人类的工作中来，那么“采用更好的算法”就相当于思考更高效的工作方式；“减少无谓的开销”则相当于减少低级重复劳动，用计算机来实现自动化。

“用空间来换时间”可能不是很容易理解。计算机即便进行重复劳动也不会有任何怨言，但还是需要人类来进行管理。如果能够将计算过一次的结果保存在某个地方，就可以缩短计算时间。

这样一来，所需要的内存空间增加了，但计算时间则可以得到缩短。在人类的工作中，应该是相当于将复杂的步骤和判断实现并总结成文档，从而提高效率的方法吧。

然而，在有限的条件下，提高工作效率的最好方法就是减负。我们所遇到的大部分工作都可以分为三种，即非得完成不可的、能完成更好但并不是必需的，以及干脆不做为好的。有趣的是，这三种工作之间的区别并非像外人所想象的那样简单。有一些工作虽然看起来绝对是必需的，但仔细想想的话就会发现也未必。

人类工作的定义比起计算机来说要更加模棱两可，像这样伴随不确定性，由惯性思维所产生的不必要不紧急的工作，如果能够砍掉的话，就能够大幅度提高工作效率。

拖延

减少不必要不紧急的工作，就能够更快地完成必要的工作，提高效率，关于这一点恐怕大家没有什么异议。不过，到底哪项工作是必要的，而哪项工作又不是必要的，要区分它们可比想象中困难得多。要找出并剔除不必要的工作，还真没那么容易。

其实，要做出明确的区分，还是有诀窍的，那就是利用人类心理上的弱点。人们普遍都有只看眼前而忽略大局的毛病，因此，当项目期限逼近时，就会产生“只要能赶上工期宁愿砸锅卖铁”这样的念头。

即便如此，估计也解决不了问题，还不如将计就计，干脆拖到不能再拖为止，这样一来，工期肯定赶不上了，只好看看哪些工作是真正必需的，剩下的那些就砍掉吧。换作平时，要想砍掉一些工作，总会有一些抵触的理由，如“这个说不定以后要用到”、“之前也一直这么做的”之类的，但在工期大限的压力面前，这些理由就完全撑不住了。这就是拖延的魔力。

不过，这个方法的副作用还是很危险的。万一估算错了时间，连必要的工作也完成不了，那可就惨了。所以说这只是个半开玩笑（但另一半可是认真的）的拖延用法，但除此之外，还有其他一些利用拖延的方法。

例如，每个任务都各自需要一定的时间才能完成，有些任务只要 5 分钟就能完成，而有些

则需要几个月。如果能够实现列出每项任务的优先级和所需的时间，就可以利用会议之间的空档等碎片时间，来完成一些较小的工作。

这种思路，和 CPU 中的乱序执行如出一辙。进一步说，对于一项任务，还可以按照“非常紧急必须马上完成的工作”、“只要不忘记，什么时候完成都可以的工作”等细分成多个子任务。

这样，按照紧急程度从高到低来完成任务的话，就可以进一步提高自己的工作效率。在这里，和“乱序执行”一样，需要注意任务之间的相互依赖关系。当相互依赖关系很强时，即使改变这些任务的顺序，也无法提高效率，这一点无论在现实的工作中还是 CPU 中都是相通的。

委派

大多数人都无法同时完成多个任务，因此可以看成是只有单一核心的硬件。即便用拖延的手段提高了工作效率，但由于同时只能处理一项任务，而每天 24 小时这个时间是固定不变的，因此一个人能完成的工作总量是存在极限的。

这种时候，“委派”就成了一个有效的手段。“委派”这个词给人的印象可能不太好，说白了，就是当以自己的能力无法处理某项任务时，转而借用他人的力量来完成的意思。如果说协作、协调、团队合作的话，大概比委派给人的印象要好一些吧。说起来，这和用多核代替单核来提升处理能力的方法如出一辙。

不过，和多核一样，这种委派的做法也会遇到一些困难。多核的困难大概有下面几种：

- ❑ 任务分割
- ❑ 通信开销
- ❑ 可靠性

这些问题，无论是编程还是现实工作中都是共通的。

如何进行妥善的任务分割是一个难题。如果将处理集中在某一个核心（或者人员）上，效率就会降低，然而要想事先对处理所需要的时间做出完全的预测也是很困难的。尤其是某项任务和其他任务有相互依赖关系的情况下，可能无论如何分割都无法提高工作效率。

我们可以把任务分为两种：存在后续处理，在任务完成时需要获得通知的“同步任务”；执行开始后不必关心其完成情况的“异步任务”。同步任务也就意味着存在依赖关系，委派的效果也就不明显。因此，如何将工作分割成异步任务就成了提高效率的关键。

在有多名成员参与的项目中，通信开销（沟通开销）也不可小觑。在人类世界中，由于“想要传达而没能传达”、“产生了误会”等原因导致的通信开销，比编程世界中要更为显著。我认为导致软件开发项目失败的最大原因，正是由于没有对这种沟通开销引起足够的重视。

最后一个问题就是“可靠性”。固然，一个人工作的话，可能会因为生病而导致工作无法进行，这也是一种风险，但随着参与项目的人数增加，成员之中有人发生问题的概率也随之上升。这就好比只有一台电脑时，往往不太担心它会出故障，但在管理数据中心里上千台的服务器时，就必须要对每天都有几台机器会出故障的情况做好准备。

当项目规模增大时，万一有人中途无法工作，就需要考虑如何修复这一问题。当然，分布式编程也是一样的道理。

非阻塞编程

在编程世界中，减负、拖延和委派是非常重要的，特别是拖延和委派恐怕还不为大家所熟悉，但今后应该会愈发成为一种重要的编程技巧。下面我们先来介绍一下在编程中最大限度利用单核的拖延方法，然后再来介绍一下运用多核的委派方法。

如果对程序运行时间进行详细分析就可以看出，大多数程序在运行时，其中一大半的时间CPU 都在无所事事。实际上，程序的一大部分运行时间都消耗在了等待输入数据等环节上，也就是说“等待”消耗了大量的 CPU 时间。

这样的等待被称为阻塞（blocking），而非阻塞编程的目的正是试图将阻塞控制在最低限度。下面我们来介绍一下作为非阻塞编程框架而备受瞩目的“node.js”。在这里，我们使用 JavaScript 来进行讲解。

node.js 框架

node.js 是一种用于 JavaScript 的事件驱动框架。提到 JavaScript，大家都知道它是一种嵌入在浏览器中、工作在客户端环境下的编程语言，而 node.js 却是在服务器端工作的。

默认嵌入到各种浏览器中的客户端语言，恐怕只有 JavaScript 这一种了，但在服务器端编程中，语言的选择则更为自由，那么为什么还要使用 JavaScript 呢？那是因为在服务器端使用JavaScript 有它特有的好处。

首先是性能。随着 Google Chrome 中 v8 引擎的出现，各大浏览器在 JavaScript 引擎性能提升方面的竞争愈演愈烈。可以说，JavaScript 是目前动态语言中处理性能最高的一种，而 node.js 也搭载了以高性能著称的 Google Chrome v8 引擎。

其次，JavaScript 的标准功能很少，这也是一个好处。和其他一些独立语言，如 Ruby 和 Python 等不同，JavaScript 原本就是作为浏览器嵌入式语言诞生的，甚至都没有提供标准的文件 I/O 等功能。

然而，在事件驱动框架上编程时，通常输入输出等可能产生的“等待”是非常麻烦的，后面我们会详细讲解这一点。node.js 所搭载的 JavaScript 引擎本身就没有提供可能会产生阻塞的功能，因此不小心造成阻塞的风险就会相应减小。当然，像死循环、异常占用 CPU 等导致的“等待”还是无法避免的。

事件驱动编程

下面我们来介绍一下，在 node.js 这样的事件驱动框架中，应该如何编程。在传统的过程型编程中，各项操作是按照预先设定好的顺序来执行的（图 1）。这与人类完成工作的一般方式相同，因此很容易理解。

相对地，在事件驱动框架所提供的事件驱动编程中，不存在事先设定好的工作顺序，而是对来自外部的“事件”作出响应，并调用与该事件相对应的“回调函数”。这里所说的事件，包括“来自外部的输入”、“到达事先设定的时间”、“发生异常情况”等情况。在事件循环框架中，主循环程序一般是用一个循环来等待事件的发生，当检测到事件发生时，找到并启动与该事件相对应的处理程序（回调函数）。当回调函数运行完毕后，再次返回到循环中，等待下一个事件（图 2）。

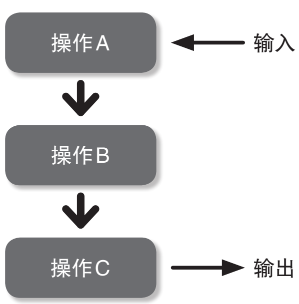


图 1　过程型编程

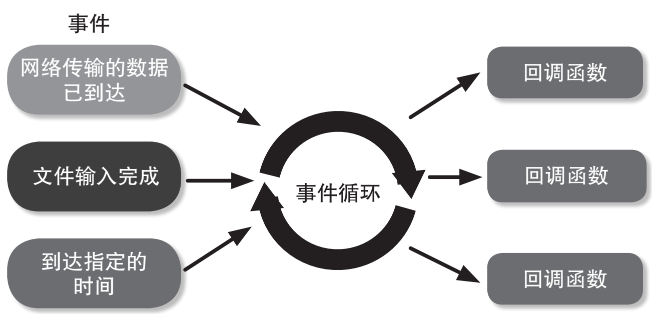


图 2　事件驱动编程

我们可以认为，过程型编程类似于每个单独的员工完成工作的方式，而事件驱动编程则类似于公司整体完成工作的方式。当发生客户下订单的事件时，销售部门（事件循环）会在接到订单后将工作转交给各业务部门（回调函数）来完成,这和事件驱动编程的模型有异曲同工之妙。

事件循环的利弊

要实现和事件循环相同的功能，除了用回调函数之外，还可以采用启动线程的方式。不过，回调只是一种普通的函数调用，相比之下，线程启动所需要的开销要大得多。而且，每个线程都需要占用一定的栈空间（Linux 中默认为每个线程 8MB 左右）。

当然，我们可以使用线程池技术，事先准备好一些线程分配给回调函数来使用，但即便如此，在资源消耗方面还是单线程方式更具有优势。此外，使用单线程方式，还不必像多线程那样进行排他处理，由此引发的问题也会相对较少。

另一方面，单线程方式也有它的缺点。虽然单线程在轻量化方面具备优势，但也同时意味着无法利用多核。此外，如果回调函数不小心产生了阻塞，就会导致事件处理的整体停滞，但在多线程 / 线程池方式中就不存在这样的问题。

node.js 编程

无论是 Debian GNU/Linux 还是我所使用的 sid[①]（开发版）中，都提供了 node.js 软件包，安装方法如下：

```
# apt-get install nodejs ↵
```

如果你所使用的发行版中没有提供软件包，就需要用源代码来安装。和其他大多数开源软件一样，node.js 的编译安装也是按 configure、make、make install 的标准步骤来进行的。

安装完毕后就可以使用 node 命令了，这是 node.js 的主体。不带任何参数启动 node 命令，就会进入下面这样的交互模式。

```
% node ↵
> console.log("hello world")
hello world
```

① sid 是 Debian 不稳定版本（开发版本）的代号：http://www.debian.org/releases/sid/。

console.log 是用于在交互模式下进行输出的函数。在非交互模式下则应该使用标准输出，因此可以认为，在正式环境下是不会使用它的。不过，如果要确认 node.js 是否工作正常，这个函数是非常方便的。在这里我们基本上是在交互模式下进行讲解的，因此会经常使用到 console.log 函数。

好，下面我们来引发一个事件试试看。要设置一个定时发生的事件及其相应的回调函数，可以使用 setTimeout() 函数。

```
> setTimeout(function(){
... console.log("hello");
... }, 2000)
```

调用 setTimeout() 的作用是，在经过第二个参数指定的时间（单位为毫秒）之后引发一个事件，并在该事件发生时调用第一个参数所指定的回调函数。Timeout 事件是一个仅发生一次的事件。

```
function () {
   console.log("hello");
}
```

这个部分是一个匿名函数，和 Ruby 中的 lambda 功能差不多。这里的重点是，调用 setTimeout() 函数之后，该函数马上就执行完毕并退出了。

setTimeout() 的作用仅仅是预约一个事件的发生，并设置一个回调函数，并没有必要进行任何等待。这与 Ruby 中的 sleep 方法是不同的，node.js 会持续对事件进行监视，基本上不会发生阻塞。

node.js 的对话模式，表面上看似乎是在一直等待标准输入，但实际上只是对标准输入传来数据这一事件进行响应，并设置一个回调函数，将输入的字符串作为 JavaScript 程序进行编译并执行。因此，在交互模式下输入 JavaScript 代码，就会被立即编译和执行，执行完毕后，会再度返回 node.js 事件循环，事件处理和对回调函数的调用，都是在事件循环中完成的。

setTimeout() 会产生一个仅发生一次的事件。如果要产生以一定间隔重复发生的事件，可以使用"setInterval()"函数来设置相应的回调函数。

```
 > var iv = setInterval(functi
on(){
 ...   console.log("hello");
 ... }, 2000)
 hello
 hello
```

通过 setInterval() 函数，我们设置了一个每 2000 毫秒发生一次的事件，并在发生事件时调用指定的回调函数。不过，每隔两秒就显示一条 hello 实在是太烦人了，我们还是把这个定期事件给取消掉吧。

为此我们需要使用 clearInterval() 函数。将 setInterval() 的返回值作为参数调用 clearInterval() 就可以解除指定的定期事件。

```
> clearInterval(iv);
```

node.js 网络编程

网络服务器才是发挥 node.js 本领的最好舞台。我们先来实现一个最简单的服务器，即将从套接字接收的数据原原本本返回的“回声服务器”。用 node.js 实现的回声服务器如图 3 所示。

作为对照，我们不用事件驱动框架，而是用 Ruby 实现另一个版本的回声服务器，如图 4 所示。

我们来连接一下试试看。在 node.js 中，要启动回声服务器，需要将图 3 中的程序保存到文件中，如 echo.js，然后执行：

```
var net = require("net");
net.createServer(function(sock){
  sock.on("data", function(data) {
    sock.write(data);
  });}).listen(8000);
```

图 3　用 node.js 实现的回声服务器

```
require "socket"

svr = TCPServer.open(8000)
socks = [svr]

loop do
  result = select(socks);
  next if result == nil
  for s in result[0]
    if s == svr
      ns = s.accept
      socks.push(ns)
    else
      if s.eof?
        s.close socks.delete(s)
      elsif str =
s.readpartial(1024)
        s.write(str)
      end
    end
  end
end
```

图 4　用 Ruby 实现的回声服务器

```
% node echo.js ↵
```

就可以启动程序了。Ruby 版则可以将图 4 的程序保存为 echo.rb，并执行：

```
% ruby echo.rb ↵
```

客户端我们就偷个懒，直接用 netcat 了。无论是使用 node.js 版还是 Ruby 版，都可以通过下列命令来连接：

```
% netcat localhost 8000 ↵
```

连接后，只要用键盘输入字符，就会得到一行和输入的字符相同的输出结果。要结束 telnet 会话，可以使用“Ctrl+C”组合键。

将图 3 程序和图 4 程序对比一下，会发现很多不同点。首先，图 4 的 Ruby 程序实际上是自行实现了相当于事件循环的部分。套接字的监听、注册、删除等管理工作也是自行完成的，这导致代码的规模变得相对较大。

node.js 版则比 Ruby 版要简洁许多。虽说采用 node.js 需要熟悉回调风格，但作为服务器的实现来说，显然还是事件驱动框架更加高效。

下面我们来详细看看 node.js 版和 Ruby 版之间的区别。

首先，node.js 版中，开头使用 require 函数引用了 net 库，net 库提供了套接字通信相关的功能。接下来调用的 net.createServer 函数，用于创建一个 TCP/IP 服务器套接字，并在该套接字上接受来自客户端的连接请求。在 createServer 的参数中所指定的函数，会被作为回调函数进行调用，当回调函数被调用时，会将客户端连接的套接字（sock）作为参数传递给它。sock 的 on 方法用于设置 sock 相关事件的回调函数。

当来自客户端的数据到达时会发生 data 事件，收到的数据会被传递给回调函数。这里我们要实现的是一个回声服务器，因此只需要将收到的数据原本返回即可。listen 方法用于在服务器套接字上监听指定的端口号。

随后，程序到达末尾，进入事件循环。需要注意的是，node.js 程序中，程序主体仅负责对事件和回调函数进行设置和初始化，实际的处理是在事件循环内完成的。

相对地，Ruby 版则需要自行管理事件循环和套接字，因此程序结构相对复杂一些。大体上是这样的：

(1) 通过 TCPSever.open 打开服务器套接字。
(2) 通过 select 等待事件。
(3) 如果是对服务器套接字产生的事件，则通过 accept 打开客户端连接套接字。
(4) 除此之外的事件，如遇到 eof?（连接结束）则关闭客户端套接字。
(5) 读取数据，并将数据原原本本回写至客户端。

在 node.js 中，上述 (2)、(3)、(4) 的部分已经嵌入在框架中，不需要以显式代码来编写，而且，程序员也不必关心资源管理等细节。正是由于这些特性，使得在回声服务器的实现上，node.js 的代码能够做到非常简洁。

node.js 回调风格

像图 3 这样将多个回调函数叠加起来的编程风格，恐怕还是要习惯一下才能上手。

下面我们通过实现一个简单的 HTTP 服务器，来仔细探讨一下回调风格。图 5 是运用 node.js 库实现的一个超简单的 HTTP 服务器。无论收到任何请求，它都只返回 hello world 这个纯文本字符串。

```
var http = require('http');

http.createServer(function(req, res) {
    res.writeHead(200, {'Content-Type':'text/plain'});
    res.write("hello world¥n");
    res.end();
}).listen(8000);
```

图 5　用 node.js 编写的 HTTP 服务器（1）

我们来实际访问一下试试看。

```
% curl http://localhost:8000/ ⏎
hello world
```

很简单吧。

我们来思考一下回调风格。图 6 所示的，是一个读取当前目录下的 index.html 文件并返回其内容的 HTTP 服务器。index.html 的读取是用 fs 库的 readFile 函数来完成的。

这个函数会在文件读取完毕后调用回调函数，也就是说即便是简单的文件输入输出也采用了回调风格。传递给回调函数的参数包括是否发生错误的信息，以及读取到的字符串。

node.js 的 fs 库中，也提供了用于同步读取操作的 readFileSync 函数，但在 node.js 中，还是推荐采用无阻塞风险的回调风格。

像这样，随着接受请求、读取文件等操作的叠加，回调函数的嵌套也会越来越深，这是回调风格所面临的一个课题。当然，我们也有方法来应对，不过关于这个问题，我们还是将来有机会再讲吧。

```
var http = require("http");
var fs = require("fs");

http.createServer(function(req, res) {
    fs.readFile("index.html", function(err, content) {
        if (err) {
          res.writeHead(404, {"Content-Type":"text/plain"});
          res.write("index.html: no such file¥n");
        }
        else {
          res.writeHead(200, {"Content-Type":"text/html; charset=utf-8"});
          res.write(content);
        }
        res.end();
    });
}).listen(8000);
```

图 6　用 node.js 编写的 HTTP 服务器（2）

node.js 的优越性

通过刚才的介绍，大家是不是能够感觉到，用 node.js 可以很容易地实现一个互联网服务器呢？即使必须要习惯 node.js 的回调风格，这样的特性也是非常诱人的。

不过，用 node.js 来实现服务器的优越性并非只有“容易”这一点而已。首先，在事件检测上，node.js 并没有采用随连接数的增长速度逐渐变慢的 select 系统调用这一传统方式，而是采用了与连接数无关，能够维持一定性能的 epoll（Linux）和 kqueue（FreeBSD）等方式。因此，在连接数上限值方面可以比较令人放心。但是，对于每个进程来说，文件描述符的数量在操作系统中是存在上限的，要缓解这一上限可能还需要一些额外的设置。

其次，node.js 的 http 库采用了 HTTP1.1 的 keep-alive 方式，来自同一客户端的连接是可以重复使用的。TCP 套接字的连接操作需要一定的开销，而通过对连接的重复使用，当反复访问同一台服务器时，就可以获得较高的性能。

再有，通过运用事件驱动模型，可以减少每个连接所消耗的资源。综合上述这些优势可以看出，同一客户端对同一服务器进行频繁连接，且连接数非常大的场景，例如网络聊天程序的实现，使用 node.js 是最合适的。

我们对图 3 的回声服务器进行些许改造，就实现了一个简单的聊天服务器（图 7）。这里所做的改造，只是将回声服务器返回输入的字符串的部分，改成了将字符串发送给当前连接的所有客户端。另外，我们还为连接中断时发生的 end 事件设置了一个回调函数，用于将客户端从连接列表中删除。

通过这样简单的修改，当多个客户端连接到服务器时，从其中任意客户端上输入的信息就可以在所有客户端上显示出来。当然，作为简易聊天程序来说，它还无法显示出某条消息是由谁发送的，因此还不是很实用。但如果能够从连接的套接字获取相关信息的话，修改起来也应该不难。

此外，我们在这里直接使用了 TCP 连接，但只要运用 keep-alive 和 Ajax（Asynchronous JavaScript and XML）技术，要用 HTTP 实现实时聊天（也就是所谓的 COMET）也并非难事。能够轻松开发出可负担如此大量连接的互联网服务器，正是 node.js 这一事件驱动框架的优势所在。

```
var net = require("net");
var clients = [];

net.createServer(function(sock){
  clients.push(sock);
  sock.on("data", function(data) {
    for (var i=0; i<clients.length; i++)
{
      clients[i].write(data);
    }
  });
  sock.on("end", function() {
    var i = clients.indexOf(sock);
    clients.splice(i, 1);
  });
}).listen(8000);
```

图 7　用 node.js 编写的网络聊天程序

EventMachine 与 Rev

当然，面向 Ruby 的事件驱动框架也是存在的，其中最具代表性的当属 EventMachine 和 Rev。EventMachine 是面向 Ruby 的事件驱动框架中的元老，实际上，在 node.js 的官方介绍中，也用了“像 EventMachine”[1]这样的说法。之所以在这里没有介绍它们，是因为相比这些框架所提供的“为每个事件启动相应的对象方法”的方式来说，node.js 这样注册回调函数的方式更加容易讲解。

① “Node is similar in design to and influenced by systems like Ruby’s Event Machine or Python’s Twisted.”（http://nodejs.org/about/）

6.5 ZeroMQ

大家还记得 6.4 节中学到的那些提高工作效率的关键词吗？就是拖延和委派。所谓拖延，就是将工作分解成细小的任务，将无法马上着手的工作拖到后面再做，从而减少等待和无用的时间，提高整体的工作效率。在 6.4 节中，我们通过 node.js 这一彻底杜绝等待的非阻塞框架，对拖延策略进行了具体的实践。

然而，无论如何削减无谓的等待，在单位时间内，每个人所能完成的工作量，或者每个 CPU 所能处理的工作量，都是存在极限的。因此，我们需要另一个策略，即委派。也就是说，将工作交给多个人，或者多个 CPU 来共同分担，从而提升整体的处理效率。

接下来我们来具体学习一下委派策略。在编程中，委派就意味着充分运用多个 CPU 来进行分布式处理。

多 CPU 的必要性

对多 CPU 系统的需求主要出于两个原因：一个是 CPU 在摩尔定律影响下的发展趋势；另一个是对绝对性能的需求。

关于前者，在摩尔定律的影响下，一直以来 CPU 性能的提升都是通过晶体管数量的增加来实现的，但随着渗漏电流等问题所形成的障碍，这种传统的性能提升方式很快就会达到极限。于是，在一块芯片上搭载多个 CPU 核心的“多核”技术便应运而生。

最近的电脑中，多个程序同时工作的多任务方式已经成为主流，因此如果 CPU 拥有多个核心能够进行并行处理，那么必然会带来直接的性能提升。美国英特尔公司推出的 Core i7 CPU 搭载了 4 个物理核心，通过超线程技术，对外能够体现 8 个核心，面向普通电脑的 CPU 能做到这样的地步，已经着实令人惊叹了。

既然普通电脑都已经配备多核 CPU 了，那么积极运用这一技术来提高工作效率也是理所当然的事。然而，和过去 CPU 本身的性能提升不同，要想在多核环境下提升处理性能，软件方面也必须要支持多核才行。

在产生对多核系统需求的两个原因中，前者属于环境问题，即“存在多核 CPU，因此想要对其进行活用”这样的态度；而后者，即对绝对处理性能的需求，可以说是一种刚性需求了。有个词叫做“信息爆炸”，也就是说，我们每天所要接触的数据量正在不断增加。各种设备通过连接到电脑和网络，在不断获取新的信息，而原本孤立存在的数据，也将通过互联网开始相互流通。

在这一变化的影响下，我们每天所要接触到的数据量正在飞速增长。既然单个 CPU 的性能提升已经遇到了瓶颈，那么通过捆绑多个 CPU 来提升性能可以说是一个必然的趋势。

无论如何，可以说，要提升处理性能，充分运用多 CPU 是毫无悬念的，为此，必须要开发出能够活用多 CPU 的软件。

阿姆达尔定律

不过，首先大家必须要记住一点，从某种意义上说也是理所当然的，那就是即便为活用多 CPU 开发了相应的软件，也不可能无限地提升工作效率。

阿姆达尔定律说的就是这件事。前面我们已经讲过，阿姆达尔定律是由吉恩·阿姆达尔（Gene Amdahl）提出的，其内容是：“（通过并行计算所获得的）系统性能提升效果，会随着无法并行的部分而产生饱和”。

能够活用多 CPU 的处理，基本上可以分解为下列步骤：

(1) 数据分割、分配
(2) 对已分配的数据进行并行处理
(3) 将已处理的数据进行集约

其中，能够并行处理的部分基本上只有 (2)，而数据的分割和集约无论有多少个 CPU，也无法最大限度地运用它们的性能。

多 CPU 的运用方法

运用多 CPU 的手段，大体上可以分成线程方式和进程方式两种。

除此之外，也有一些支持分布式编程的框架，但从内部原理来看，还是采用了线程、进程两种方式中的一种。

线程和进程都是多 CPU 的有效运用手段，但各自拥有不同的性质，也拥有各自的优点和缺点，应该根据应用程序的性质进行选择。

线程是在一个进程中工作的控制流程，其最大的特点是，工作在同一个进程内的多个线程之间，其内存空间是共享的。这一特点可以说是喜忧参半，却决定了线程的性格。

共享内存空间，就意味着在线程间操作数据时不需要进行复制。尤其是在线程间需要操作的数据非常多的情况下，像这样无需复制就能够传递数据，在处理性能方面是非常有利的。

然而，在获得上述好处的同时，也会带来一定的隐患。共享内存空间，也意味着某个线程中操作的数据结构，可能被其他线程修改。由于各线程是独立工作的，因此可能会导致一些特殊时机才出现的、非常难以发现的 bug。

虽然大多数情况下不会出问题，然而在非常偶然的情况下，两个线程同时访问同一数据结构，就会导致程序崩溃，而且这样的 bug 是很难重现的。一想到可能要去寻找这样的 bug，就会不由得感到眼前发黑。

此外，线程是在一个进程中工作的控制流程，反过来说，所有的处理都必须在同一个进程中完成，这也就意味着，如果要只采用线程方式来运用多 CPU，就必须在一台电脑上完成所有的处理。

然而，即便是在多核已经司空见惯的现在，一台电脑上能够使用的核心数量最多也就是 4 核，算上超线程也就是 8 核。服务器的话可能会配备更多的核心。但无论如何，现在还无法达到数百甚至数千核心的规模。如果要实现更高的并行度，仅靠线程还是会遇到极限的。

相对地，多进程方式同样是喜忧参半的，其特点正好和线程方式相反，即无法共享内存空间，但处理不会被局限在一台计算机上完成。

无法共享内存空间，就意味着在操作数据时需要进行复制操作，对性能有不利影响。但是，在并发编程中，数据共享一向是引发问题的罪魁祸首，因此从牺牲性能换取安全性的角度来说，也可以算是一个优点。

此外，刚才也提到过，在一台计算机所搭载的 CPU 数量不多的情况下，使用进程方式能够通过多台计算机构成的系统运用更多的 CPU 进行并行处理，这是一个很大的优点。不过，在这样的场景中，选择何种手段实现进程间的数据交换就显得非常重要。

尽管个人喜好可能并不可靠，但相比线程方式来说，我更加倾向于使用进程方式。

理由有两个。首先，要安全使用线程相当困难。相比共享内存带来的性能提升来说，由于

状态的共享会导致一些偶发性 bug，因此风险大于好处。

其次，对性能提升带来的贡献，会受到该计算机中搭载的 CPU 核心数量上限的制约，因此其可扩展性相对较低。换句话说，可能搞得很辛苦，却得不到太多的好处，性价比不高。

当然，在某些情况下，线程方式比进程方式更合适，并不是说线程方式就该从世界上消失了。不过，我认为线程方式只应该用在有限的情况中，而且是用在一般用户看不见的地方，而不应该在应用程序架构模型的尺度上使用。当然，我知道一定有人不同意我的看法。

进程间通信

由于线程是共享内存空间的，因此不会发生所谓的通信。但反过来说，则存在如何防止多个进程同时访问数据的排他控制问题。

相对地，由于进程之间不共享数据，因此需要显式地进行通信。进程间通信的手段有很多种，其中具有代表性的有下列几种。

- 管道
- SysV IPC
- TCP 套接字
- UDP 套接字
- UNIX 套接字

下面我们来分别简单介绍一下。

管道

所谓管道，就是能够从一侧输出，然后从另一侧读取的文件描述符对。Shell 中的管道等也是通过这一方式实现的。

文件描述符在每个进程中是独立存在的，但创建子进程时会继承父进程中所有的文件描述符，因此它可以用于在具有父子、兄弟关系的进程之间进行通信。

例如，在具有父子关系的进程之间进行管道通信时，可以按下列步骤操作。在这里为了简单起见，我们只由子进程向父进程进行通信。

① 首先，使用 pipe 系统调用，创建一对文件描述符。下面我们将读取一方的文件描述符称为“r”，将写入一侧的文件描述符称为“w”。
② 通过 fork 系统调用创建子进程。
③ 在父进程一方将描述符 w 关闭。
④ 在子进程一方将描述符 r 关闭。
⑤ 在子进程一方将要发送给父进程的数据写入描述符 w。
⑥ 在父进程一方从描述符 r 中读取数据。

为了实现进程间的双向通信，需要按与上述相同的步骤创建两组管道。虽然比较麻烦，但难度不大。

和 Shell 一样，要在两个子进程之间进行通信，只要创建管道并分配给各子进程，各子进程之间就可以直接通信了。为了将进程与进程联系起来，每次都需要执行上述步骤，一旦自己亲自尝试过一次之后，就会明白 Shell 有多么强大了。

管道通信只能用于具有父子、兄弟关系、可共享文件描述符的进程之间，因此只能实现同一台电脑上的进程间通信。实际上，如果使用后面要介绍的 UNIX 套接字，就可以在不具有父子关系的进程之间传递文件描述符，但只能用在同一台电脑上的这一限制依然存在。

SysV IPC

UNIX 的 System V（Five）版本引入了一组称为 SysV IPC 的进程间通信 API，其中 IPC 就是 Inter Process Communication（进程间通信）的缩写。

SysV IPC 包括下列 3 种通信方式。

- 消息队列
- 信号量
- 共享内存

消息队列是一种用于进程间数据通信的手段。管道只是一种流机制，每次写入数据的长度等信息是无法保存的，相对地，消息队列则可以保存写入消息的长度。

信号量（semaphore）是一种带有互斥计数器的标志（flag）。这个词原本是荷兰语“旗语”的意思，在信号量中可以设定对某种“资源”同时访问数量的上限。

共享内存是一块在进程间共享的内存空间。通过将共享内存空间分配到自身进程内存空间

中（attach）的方式来访问。由于对共享内存的访问并没有进行排他控制，因此无法避免一些偶发性问题，必须使用信号量等手段进行保护。

不过，说实话，我自己从来没用过 SysV IPC。原因有很多，最重要的一个原因是资源泄漏。由于 SysV IPC 的通信路径能够跨进程访问，因此在使用时需要向操作系统申请分配才能进行通信，通信完全结束之后还必须显式销毁，如果忘记销毁的话，就会在操作系统内存中留下垃圾。相比之下，管道之类的方式，在其所属进程结束的同时会自动销毁，因此比 SysV IPC 要更加易用。

其次，学习使用新的 API 要花一些精力，但结果也只能用在一台电脑上的进程间通信中，真是让人没什么动力去用呢。

最后一个原因，就是在 20 多年前我开始学习 UNIX 编程的时候，并非所有的操作系统都提供了这一功能。当时，擅长商用领域的 AT&T 系 System V UNIX 和加州大学伯克利分校开发的 BSD UNIX 正处于对峙时期。那个时候，我主要用的是 BSD 系 UNIX，而这个系统就不支持 SysV IPC。现在大多数 UNIX 系操作系统，包括 Linux，都支持 SysV IPC 了，但过去则并非如此，也许正是这种历史原因造成我一直都没有去接触它。

后来，System V 与 BSD 之间的对峙，随着双方开始吸收对方的功能而逐步淡化，再往后，严格来说，不属于 System V 和 BSD 两大阵营的 Linux 成为了 UNIX 系操作系统的最大势力，而曾经的对峙也成为了历史，这个结局恐怕在当时是谁都无法想象的吧。

说到底，用都没用过的东西要给大家介绍实在是难上加难。关于 SysV IPC 的用法，大家可以在 Linux 中参考一下：

```
# man svipc ⏎
```

其他操作系统中，也可以从创建消息队列的 msgget 系统调用的 man 页面中找到相关信息。

套接字

System V 所提供的进程间通信手段是 SysV IPC，相对地，BSD 则提供了套接字的方式。和其他进程间通信方式相比，套接字有一些优点。

- 通信对象不仅限于同一台计算机，或者说套接字本身主要就是为计算机间的通信而设计的。
- （和 SysV IPC 不同）套接字也是一种文件描述符，可进行一般的输入输出。尤其是可以使用 select 系统调用，在通常 I/O 的同时进行“等待”，这一点非常方便。

- ❑ 套接字在进程结束后会由操作系统自动释放，因此无需担心资源泄漏的问题。
- ❑ 套接字（由于其优秀的设计）从很早开始就被吸收进 System V 等系统了，因此在可移植性方面的顾虑较少。

现代网络几乎完全依赖于套接字。各位所使用的几乎所有服务的通信都是基于套接字实现的，这样说应该没有什么大问题。

套接字分为很多种，其中具有代表性的包括：

- ❑ TCP 套接字
- ❑ UDP 套接字
- ❑ UNIX 套接字

TCP（Transmission Control Protocol，传输控制协议）套接字和 UDP（User Datagram Protocol，用户数据报协议）套接字都是建立在 IP（Internet Protocol，网际协议）协议之上的上层网络通信套接字。这两种套接字都可用于以网络为媒介的计算机间通信，但它们在性质上有一些区别。

TCP 套接字是一种基于连接的、具备可靠性的数据流通信套接字。所谓基于连接，是指通信的双方是固定的；而所谓具备可靠性，是指能够侦测数据发送成功或是发送失败（出错）的状态。

所谓数据流通信，是指发送的数据是作为字节流来处理的，和通常的输入输出一样，不会保存写入的数据长度信息。

看了上面的内容，大家可能觉得这些都是理所当然的嘛。我们和 UDP 套接字对比一下，就能够理解其中的区别了。

UDP 套接字和 TCP 套接字相反，是一种能够无需连接进行通信、但不具备可靠性的数据报通信套接字。所谓能够无需连接进行通信，是指无需固定连接到指定对象，可以直接发送数据；不具备可靠性，是指可能会出现中途由于网络状况等因素导致发送数据丢失的情况。

在数据报通信中，发送的数据在原则上是能够保存其长度的。但是，在数据过长等情况下，发送的数据可能会被分割。

先不说无连接通信这一点，UDP 的其他一些性质可能会让大家感到非常难用。这是因为 UDP 几乎是原原本本直接使用了作为其基础的 IP 协议。相反，TCP 为了维持可靠性，在 IP 协议之上构建了各种机制。UDP 的特点是结构简单，对系统产生的负荷也较小。

因此，在语音通信（如 IP 电话等）中一般会使用 UDP，因为通信性能比数据传输的可靠性

要更加重要，也就是说，相比通话中包含少许杂音来说，还是保证较小的通话延迟要更加重要。

TCP 套接字和 UDP 套接字都是通过 IP 地址和端口号来进行工作的。例如，http 协议中的“http://www.rubyist.net:80/”就表示与 www.rubyist.net（2012 年 3 月 27 日当时的 IP 地址为 221.186.184.67）所代表的计算机的 80 号端口建立连接。

UNIX 套接字

同样是套接字，UNIX 套接字和 TCP、UDP 套接字相比，可以算是一个异类。基于 IP 的套接字一般是通过主机名和端口号来识别通信对象的，而 UNIX 套接字则是在 UNIX 文件系统上创建一个特殊文件，并用该文件的路径进行识别。由于这种方式使用的是文件系统，因此大家可以看出，UNIX 套接字只能用于同一台计算机上的进程间通信。

UNIX 套接字并不是基于 IP 的套接字，它可用于向同一台计算机上其他进程提供服务的某种服务程序。例如有一种叫做 canna 的汉字转换服务，就是通过 UNIX 套接字来接受客户端连接的。

ZeroMQ

在进程间通信手段中，套接字算是非常好用的，但即便如此，在考虑对工作进行“委派”时，其易用性还并不理想。套接字本来是为网络服务器的实现而设计的，但作为构建分布式应用程序的手段来说，却显得有些过于原始了。

ZeroMQ 就是为了解决这一问题而诞生的，它是一种为分布式应用程序开发提供进程间通信功能的库。

ZeroMQ 的特点在于灵活的通信手段和丰富的连接模型，并且它可以在 Linux、Mac OS X、Windows 等多种操作系统上工作，也支持由多种语言进行访问。ZeroMQ 所支持的语言列表如表 1 所示。

表1　ZeroMQ支持的语言一览

Ada	Lua	CommonLisp	Perl
BASIC	Node.js	Erlang	Python
C	Objective-C	Go	Racket
C#	ooc	Haskell	Ruby
C++	PHP	Java	Scala

ZeroMQ 提供了下列底层通信手段。无论使用哪种手段，都可以通过统一的 API 进行访问，这一点可以说是 ZeroMQ 的魅力。

- ❑ tcp
- ❑ ipc
- ❑ inproc
- ❑ multicast

tcp 就是 TCP 套接字，它使用主机名和端口号进行连接。根据 TCP 套接字的性质，从其他计算机也可以进行连接，但由于 ZeroMQ 不存在身份认证这样的安全机制，因此建议大家不要在互联网上公布 ZeroMQ 的端口号。

ipc 用于在同一台计算机上进行进程间通信，使用文件路径来进行连接。实际通信中使用何种方式与实现有关，在 UNIX 系操作系统上采用的是 UNIX 套接字，在 Windows 上也许是用一般套接字来通信的吧。

inproc 用于同一进程中的线程间通信。由于线程之间是共享内存空间的，因此这种通信方式是无需复制的。使用 inproc 通信，可以在活用线程的同时，避免麻烦的数据共享，不仅通信效率高，编写的程序也比较易读。

multicast 是一种采用 UDP 实现的多播通信。为了实现一对多的通信，如果使用一对一的 TCP 方式，则需要对多个对象的 TCP 连接反复进行通信，但如果使用原本就用于多播通信的 multicast，就可以避免无谓的重复操作。

不过，UDP 通信，尤其是多播传输，在一些路由器上是被禁止的，因此这种方式并不能所向披靡，这的确是个难点。

ZeroMQ 的连接模型

ZeroMQ 为分布式应用程序的构建提供了丰富多彩的连接模型，主要有以下这些。

- ❑ REQ/REP
- ❑ PUB/SUB
- ❑ PUSH/PULL
- ❑ PAIR

REQ/REP 是 REQUEST/REPLY 的缩写，表示向服务器发出请求（request），服务器向客户

端返回应答（reply）这样的连接模型（图 1）。

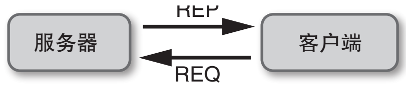


图 1　REQ/REP 模型

作为网络连接来说，这种方式是非常常见的。例如 HTTP 等协议，就遵循 REQ/REP 模型。通过网络进行函数调用的 RPC（Remote Procedure Call，远程过程调用）也属于这一类。REQ/REP 是一种双向通信。

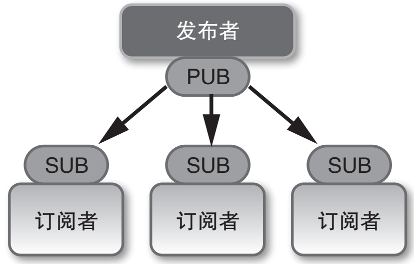


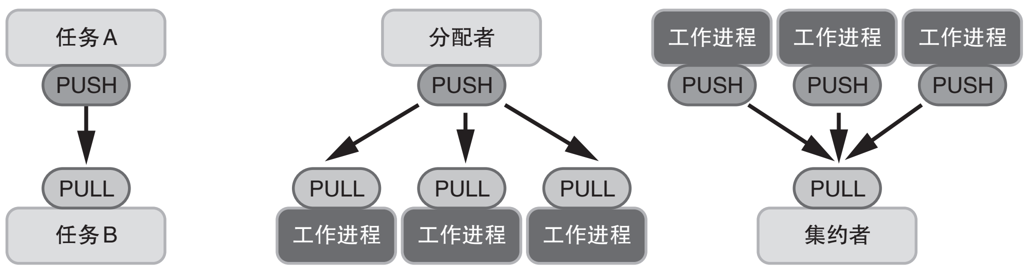
图 2　PUB/SUB 模型

PUB/SUB 是 PUBLISH/SUBSCRIBE 的缩写，即服务器发布（publish）信息时，在该服务器上注册（subscribe，订阅）过的客户端都会收到该信息（图 2）。这种模型在需要向大量客户端一起发送通知，以及数据分发部署等场合非常方便。PUB/SUB 是一种单向通信。

PUSH/PULL 是向队列中添加和取出信息的一种模型。PUSH/PULL 模型的应用范围很广，如果只有一个数据添加方和一个数据获取方的话，可以用类似 UNIX 管道的方式来使用（图 3a），如果是由一台服务器 PUSH 信息，而由多台客户端来 PULL 的话，则可以用类似任务队列的方式来使用（图 3b）。

在图 3b 的场景中，处于等待状态的任务中只有一个能够取得数据。相对地，PUB/SUB 模型中则是所有等待的进程都能够取得数据。

反过来说，如果有多个进程来 PUSH，则能够用来对结果进行集约（图 3c）。和 PUB/SUB 一样，PUSH/PULL 也是一种单向通信。



(a) 管道（一对一）　(b) 任务分配（一对多）　(c) 任务集约（多对一）

图 3　PUSH/PULL 模型

PAIR 是一种一对一的双向通信。说实话，在我所了解的范围内，还不清楚这种模型应该如何使用。

ZeroMQ 的安装

首先安装 ZeroMQ 库的主体。在 Debian 中提供的软件包叫做 libzmq-dev，安装方法如下：

```
$ apt-get install libzmq-dev ↵
```

如果在你所使用的平台上没有提供二进制软件包，也可以从 http://www.zeromq.org/ 下载源代码进行编译安装。截止到 2012 年 3 月 27 日，其最新版本为 2.1。

ZeroMQ 的标准 API 是以 C 语言方式提供的，但 C 语言实在太繁琐了，因此这里的示例程序我们用 Ruby 来编写。Ruby 的 ZeroMQ 库叫做 zmq，可以通过 RubyGems 进行安装。在安装 ZeroMQ 基础库之后，运行

```
$ gem install zmq ↵
```

即可安装 ZeroMQ 的 Ruby 库。

ZeroMQ 示例程序

首先，我们来看看最简单的 REQ/REP 方式。图 4 是用 Ruby 编写的 REQ/REP 服务器。在这里我们只接受来自本地端口的请求，如果将 127.0.0.1 的部分替换成“*”就可以接受来自任何主机的请求了。客户端程序如图 5 所示。

```
require 'zmq'

context = ZMQ::Context.new
socket = context.socket(ZMQ::REP)
socket.bind("tcp://127.0.0.1:5000")

loop do
  msg = socket.recv
  print "Got ", msg, "\n"
  socket.send(msg)
end
```

图 4　ZeroMQ REQ/REP 服务器

```
require 'zmq'

context = ZMQ::Context.new
socket = context.socket(ZMQ::REQ)
socket.connect("tcp://127.0.0.1:5000")

for i in 1..10
  msg = "msg %s" % i
  socket.send(msg)
  print "Sending ", msg, "\n"
  msg_in = socket.recv
  print "Received ", msg, "\n"
end
```

图 5　ZeroMQ REQ/REP 客户端

这样我们就完成了一个万能 echo 服务器及其相应的客户端。

ZeroMQ 可以发送和接收任何二进制数据，如果我们发送 JSON 和 MessagePack 字符串的话，就可以轻松实现一种 RPC 的功能。

要进行通信，可以按顺序启动图 4 和图 5 的程序。有意思的是，一般的套接字程序中，必须先启动服务器，但 ZeroMQ 程序中，先启动客户端也是可以的。

ZeroMQ 是按需连接的，因此当连接对象尚未初始化时，客户端会进入待机状态。启动顺序自由这一点非常方便，尤其是 PUB/SUB 和 PUSH/PULL 模型的连接中，如果所有的服务器和客户端只能按照一定顺序来启动，那制约就太大了，而 ZeroMQ 则可以将我们从这样的制约中解放出来。

此外，ZeroMQ 还可以同时连接多个服务器。如果在图 5 程序的第 5 行，即 connect 那一行之后，再添加一行相同的语句（例如只改变端口号），就可以对两个服务器交替发送请求。通过这样的方式，可以很容易实现负载的分配。

下面我们再来看看用 PUSH/PULL、PUB/SUB 模型实现的一个简单的聊天程序。这个示例由 3 个程序构成。程序 1（图 6）是聊天发言用的程序。通过将命令行中输入的字符 PUSH 给服务器来“发言”。程序 2（图 7）是显示发言用的程序。通过 SUBSCRIBE 的方式来获取服务器 PUBLISH 的发言信息，并显示在屏幕上。实际的聊天系统中，客户端应该是由程序 1 和程序 2 结合而成的。

```
require 'zmq'

context = ZMQ::Context.new
socket = context.socket(ZMQ::PUSH)
socket.connect("tcp://127.0.0.1:7900")

socket.send(ARGV[0])
```

图 6　聊天发言程序

程序 3（图 8）是聊天服务器，它通过 PULL 来接收发言数据，并将其原原本本 PUBLISH 出去，凡是 SUBCRIBE 到该服务器的客户端，都可以收到发言内容（图 9）。无论有多少个客户端连接到服务器，ZeroMQ 都会自动进行管理，因此程序的实现就会比较简洁。

```
require 'zmq'

context = ZMQ::Context.new
socket = context.socket(ZMQ::SUB)
socket.connect("tcp://127.0.0.1:7901")
# 显示执行SUBSCRIBE操作并对消息进行取舍选择
# 空字符串表示全部获取的意思
socket.setsockopt(ZMQ::SUBSCRIBE, "")

loop do
  puts socket.recv
end
```

图 7　聊天显示程序

```
require 'zmq'

context = ZMQ::Context.new
receiver = context.socket(ZMQ::PULL)
receiver.bind("tcp://127.0.0.1:7900")
clients = context.socket(ZMQ::PUB)
clients.bind("tcp://127.0.0.1:7901")

loop do
  msg = receiver.recv
  printf "Got %s¥n", msg
  clients.send(msg)
end
```

图 8　聊天服务器程序

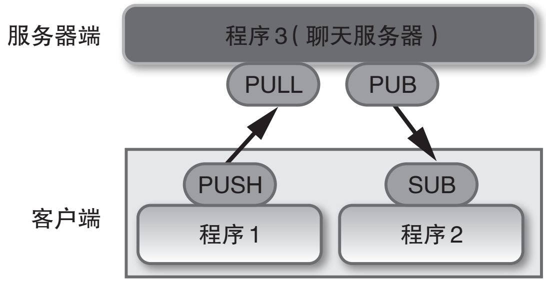


图9 聊天程序的工作方式

小结

ZeroMQ 是一个用简单的 API 实现进程间通信的库。和直接使用套接字相比，它在一对多、多对多通信的实现上比较容易。在对多 CPU 的运用中，横跨多台计算机的多进程间通信是不可或缺的，因此在需要考虑可扩展性的软件开发项目中，像 ZeroMQ 这样的进程间通信库，今后应该会变得越来越重要。

“多核时代的编程”后记

有一句话在这本书中说过很多次，各位读者可能也已经听腻了，不过在这里我还是想再说一次：现在是多核时代。

所谓多核，原本是指在一块芯片上封装多个 CPU 核心的意思。截至 2012 年 4 月，一般能够买到的电脑基本上都搭载了 Intel Core i5 等多核 CPU 芯片，这一事实也是这个时代的写照。

本书中所说的多核，并不单指多核 CPU 的使用，大多数情况下指的是“运行一个软件系统可以利用多个 CPU 核心”这个意思。在这样的场景中，并不局限于一块芯片。由多块芯片，甚至是多台计算机组成的环境，也可以看作是多核。按照这样的理解，云计算环境可以说是一种典型的多核环境吧。

多核环境中编程的共同点在于，在传统的编程风格中，程序是顺序执行的，因此只能用到单独一个核心。而要充分发挥多核的优势，就必须通过某些方法，积极运用多个 CPU 的处理能力。

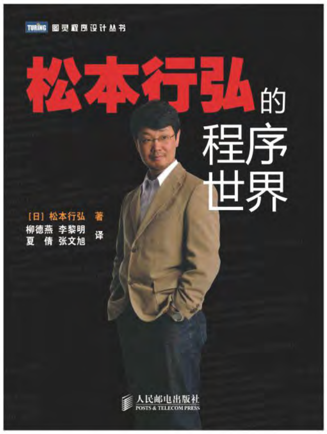
本书中介绍了一些活用多个 CPU 的方法，包括 UNIX 进程的活用、通过异步 I/O 实现并行化、消息队列等，这些都是非常有前途的技术。然而，UNIX 进程（在基本的使用方法中）只能用在一台计算机中；而异步 I/O 虽然能提高效率，但其本身无法运用多核；消息队列目前也没有强大到能够支持数百、数千节点规模系统的构建。

从超级计算机的现状进行推测，在不远的将来，云计算环境中的“多核系统”就能够达到数万节点、数十万核心的规模。要构建这样的系统，用现在的技术是可以实现的，但并非易事。

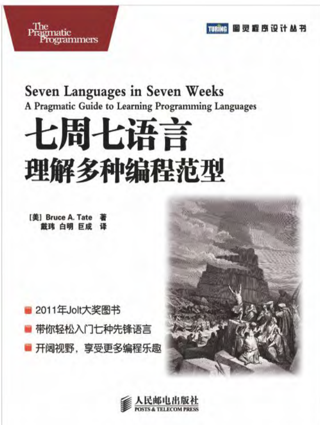
因此，在这一方面，今后还需要更大的进步。

版权声明

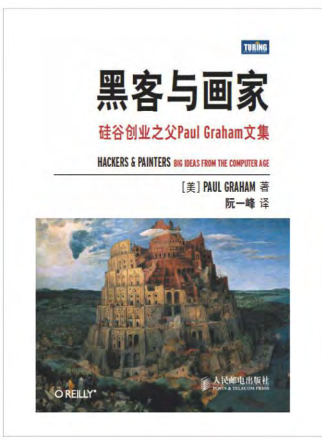
图灵好书推荐



成为一流程序员的14种思考术



2011年Jolt大奖图书



跟随硅谷创业之父，走进黑客的世界



代码的未来

The Future of Computing

站在摩尔定律的尽头，

重新理解编程的本质。

图灵社区：www.ituring.com.cn
图灵微博：@图灵教育 @图灵社区
反馈/投稿/推荐邮箱：contact@turingbook.com
热线：(010) 51095186 转 604

分类建议 计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-31751-3

ISBN 978-7-115-31751-3

定价：79.00元

欢迎加入

图灵社区

电子书发售平台

电子出版的时代已经来临，在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版的梦想。你可以联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者，这极大地降低了出版的门槛。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果有意翻译哪本图书，欢迎来社区申请。只要通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

读者交流平台

在图灵社区，读者可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。欢迎大家积极参与社区开展的访谈、审读、评选等多种活动，赢取银子，可以换书哦！